ampus omputing etwork

(12)

"AN IMPLEMENTATION OF THE MSG
INTERPROCESS COMMUNICATION PROTOCOL"

Semiannual Technical Report
July 1976 - December 1976

CCN/TR12

L. P. Rivas

H. C. Ludlam

R. T. Braden

niversity of alifornia, os ngeles

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CCN/TR12
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): An Implementation of the MSG Interprocess Communication Protocol,

5. TYPE OF REPORT & PERIOD COVERED: Semi-annual Technical 7/1/76 - 12/31/76

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): L. P. Rivas, H. C. Ludlam, R. T. Braden

8. CONTRACT OR GRANT NUMBER(s): MDA 903-74-C-0083, ARPA Order 2543

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Campus Computing Network
UCLA C0012
Los Angeles, California 90024

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
Program Code 4P10
ARPA Order No. 2543/3

11. CONTROLLING OFFICE NAME AND ADDRESS:
Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

12. REPORT DATE: May 1977

13. NUMBER OF PAGES: 128

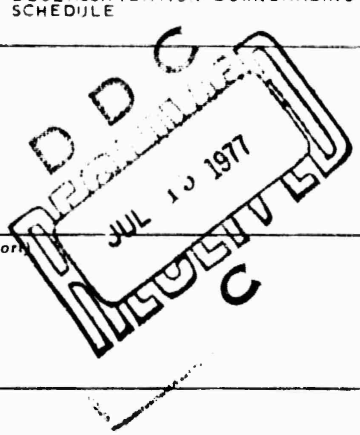
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): None

15a. DECLASSIFICATION DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution unlimited.



17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

National Software Works, ARPANET, protocol, MSG, interprocess communications, message, generic addressing, access method, process names, event signals, connection, buffering, flow control, interactive debugging, process creation, configuration, time-out, program logic, Exchange protocol

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report covers technical development at UCLA-CCN relating to the National Software Works (NSW), during the period July 1, 1976, through December 31, 1976, and is specifically concerned with the CCN implementation of MSG, the interprocess communication mechanism for the NSW. MSG is a message-oriented protocol which also provides subsidiary connection-oriented facilities.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014 6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract

PL/MSG, the process-callable interface to CCN's MSG, is an access method designed particularly for use by PL/I programs. PL/MSG communicates with the central message-switching mechanism M*S*G, which executes within the NCP job. Section 2 defines the interface which PL/MSG presents to a process, while Section 3 describes MSGBUG, an interactive debugging mechanism built into PL/MSG. These two sections are intended for those who implement processes which will use CCN's MSG for communication.

The later sections of the report describe the internal design and logic of the CCN implementation, in increasing detail and specificity. Section 4 describes the operating algorithms and the configuration tables for the CCN implementation, without giving programming details. Section 5 presents a general view of the internal organization and Exchange protocols used by M*S*G, and can be understood with little knowledge of OS/MVT or the CCN system. Finally, Section 6 details the data structures, program logic, and system interfaces of M*S*G.

CAMPUS COMPUTING NETWORK

University of California at Los Angeles
405 Hilgard Avenue,
Los Angeles, California 90024

"An Implementation of the MSG Interprocess Communication Protocol"

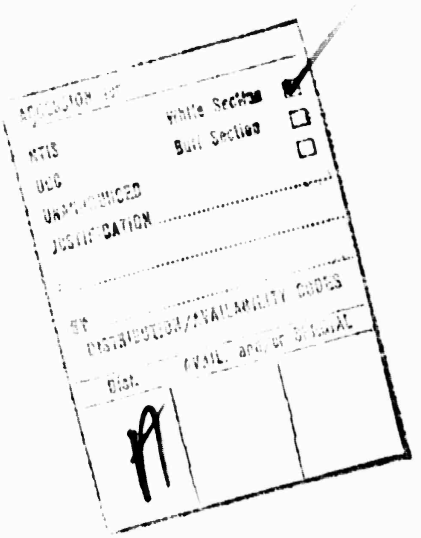
Semiannual Technical Report
July 1976 - December 1976

CCN/TR12

L. P. Rivas

H. C. Ludlam

R. T. Braden



This work was sponsored by the Advanced Research Projects Agency of the Department of Defense, under Contract Number MDA903 74C 0083, Order 2543/3.

REPORT SUMMARY

This report covers technical development at UCLA-CCN relating to the National Software Works (NSW), during the period July 1, 1976 through December 31, 1976. The NSW is a distributed multi-computer operating system based on the ARPANET. The primary goal of the NSW project at CCN is to make the CCN IBM 360/91 a "tool-bearing host" within the NSW.

This report is specifically concerned with the CCN implementation of MSG, the interprocess communication mechanism for the NSW. MSG is basically message-oriented but also provides subsidiary connection-oriented facilities.

PL/MSG, the process-callable interface in CCN's MSG implementation, is an access method designed particularly for use by PL/I programs. PL/MSG communicates with the central message-switching mechanism M*S*G, which executes within the NCP job. Section 2 of this report defines the interface which PL/MSG presents to a process, while Section 3 describes MSGBUG, an interactive debugging mechanism built into PL/MSG. These two sections are intended for those who implement processes which will use CCN's MSG for communication.

The later sections of the report describe the internal design and logic of the CCN implementation, in increasing detail and specificity. Section 4 describes the operating algorithms and the configuration tables for M*S*G, without giving programming details. Section 5 presents a general view of the internal organization and Exchange protocols used by M*S*G, and can be understood with little knowledge of OS/MVT or the CCN system. Finally, Section 6 details the data structures, program logic, and system interfaces of M*S*G.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of tne Defense Advanced Research Projects Agency or of the United States Government.

ILLUSTRATIONS

1. INTRODUCTION

1.1. The NSW and CCN

The UCLA Campus Computing Network (CCN) is one of the participating organizations in an R&D effort to build the "National Software Works" or "NSW" [1,2]. The NSW is designed to be a computer software system to support large-scale software production efforts. It will give users convenient and controlled access to software development "tools" available on diverse host computers linked through a network, and will create a global file system spanning these hosts. The prototype NSW under construction is based on the ARPANET.

The mechanism of the NSW will essentially be a distributed multi-computer operating system. An NSW user will log into the NSW and may then execute "tools" by name, generally without knowing which server host is actually executing a particular tool. To select a file for the tool, the user will enter the NSW file name. The NSW file mechanism will automatically locate a suitable physical copy of that file and copy it, across the ARPANET if necessary, into the local file-space in which the tool is executing.

The principal components of the NSW are:

FE -- Front End

The Front End program is the interface to the user terminal. It may be a separate process in a time-shared computer system (e.g., the PDP-10 Tenex system), or it may be a mini-host dedicated to terminal access for users of the NSW.

WM -- Works Manager

The Works Manager, the central controlling and record-keeping mechanism of the NSW [3], performs many operating system functions for the NSW server hosts. The WM maintains a data base containing the user profiles and access rights, the file catalog, and the tool descriptors. In addition, it keeps dynamic tables of the currently-active users and tools.

Functionally, the WM is organized into a set of subroutines or "procedures" which are "called" by the other components of NSW. The WM in turn may call procedures in either the FE or the server hosts.

TBH-- Tool-Bearing Host

In the NSW, a server host is called a "Tool-Bearing Host", or "TBH".

The role of the UCLA Campus Computing Network in the NSW development is to make CCN's IBM 360/91 a Tool-Bearing Host.

Initial TBH software development at CCN [4,5] resulted in the implementation of a facility to execute "batch tools", i.e., tools which can be executed in background using the normal batch-processing machinery of the 360/91. More recently, work at CCN has focused on the software components necessary for interactive tools under TSO. The necessary components are:

* An MSG Server

MSG is the interprocess communication protocol for the NSW [6]. MSG is oriented towards messages rather than connections, with each message carrying its destination process address.

* A File Package ("FP")

The File Package [7] is invoked directly by the Works Manager or indirectly by another File Package to perform NSW file operations -- copying files locally or across the ARPANET, converting their representation to/from a form suitable for a particular tool, and deleting files.

* A Foreman ("FM")

The Foreman [8] is logically the extension of the local operating system required for NSW; it is also the interface between the Works Manager and the tools.

This Technical Report is devoted in particular to the CCN implementation of the NSW interprocess communication mechanism MSG. The implementation described here is for an IBM System 360/370 computer running the CCN Network Control Program under the IBM operating system OS/MVT.

The remainder of this introduction discusses the MSG protocol and the general objectives and design decisions in the CCN implementation. Section 2, describing the process-callable interface to the CCN MSG, is intended for a programmer writing an NSW process. Section 3 describes MSGBUG, an interactive debugging tool for processes using MSG.

Section 4 gives the information necessary for installing and maintaining the configuration tables required by CCN's MSG.

Section 5 presents an overview of the internal organization of CCN's MSG; it should not require a knowledge of OS/MVT or of the details of CCN's NCP for understanding. Finally, Section 6 is a complete explanation of the internal logic of M*S*G, the central controller of CCN's MSG, suitable for systems programmers who must install, maintain, or modify the MSG programs.

The design of the CCN implementation was primarily performed by L. P. Rivas and H. C. Ludlam of the CCN staff. Rivas accomplished the entire implementation of M*S*G, while Ludlam designed and coded MSGBUG. The process-callable interface PL/MSG was a joint effort.

1.2. The MSG Protocol and the Exchange

The MSG protocol for communication among NSW processes was designed by a committee composed of: Robert Thomas, Richard Schantz, and Paul Johnson of Bolt, Beranek, and Newman, Inc., and Stuart Schaffner and Robert Millstein of Massachusetts Computer Associates, Inc. [6]

The abstract definition of MSG is independent of any particular medium of communication between host computers. However, the MSG protocol is being implemented on the ARPANET for the NSW. From the ARPANET viewpoint, MSG is a "third-level" protocol, built upon the standard host-to-host protocol [11].

MSG provides three different mechanisms which two "user" processes on the same or different hosts may use to communicate with each other:

* Messages
* Alarms (i.e., software interrupts)
* Direct connections.

In each host supporting MSG, there must be an MSG cont. program to interpret process addresses and to queue messages and alarms for remote and local processes. In the CCN implementation, we call this control program "M*S*G". It is conceptually unimportant whether M*S*G is implemented by a single explicit process, by a set of interrelated processes, or as an integral part of the "hard-core" Supervisor.

M*S*G must multiplex messages to/from the local user processes into the single Network connection pair to a particular remote MSG host. Thus, M*S*G must issue ARPANET Initial Connection Protocol (ICP) [12] sequences when necessary to connect to MSG control programs on other hosts, and respond to such sequences initiated by other hosts. An MSG "user" process, i.e., a process in the host that uses the MSG facility, must communicate through its local M*S*G control program.

We note that this organization introduces an additional requirement for interprocess communication, internal to each NSW host -- communication between the user processes and the MSG controller M*S*G. Presumably, each host has an internal IPC (interprocess communication) mechanism for this purpose. The IPC mechanism used on the IBM 360/91 at CCN is called "the Exchange" [9,10].

For a complete definition of MSG, the reader is referred to [6]. However, it is interesting and useful to briefly review here the features of MSG, and to compare them with CCN's Exchange.

The significant features of the MSG protocol are as follows:

(1) Individually-addressed messages

Each MSG message and alarm carries its own destination address. Thus, MSG is message-oriented rather than connection-oriented like Exchange or the ARPANET host-to-host protocol [11]. For occasional or brief communication, the message orientation is clearly simpler for the communicating processes than opening connections would be.

(2) Generically-addressed messages

MSG uses a "generic address" to establish initial contact between two processes; thereafter the processes must use specific addresses to continue the conversation. This generic address is analogous to the symbolic "tag" used by the Exchange for opening connections, and the specific MSG address essentially plays the role of the Exchange window handle.

(3) Implicit Opening of Connections

Although MSG deals with individually-addressed messages while Exchange is connection-oriented, each requires the first operation in a conversation to be different from subsequent message transmissions. In MSG, a logical path:

(process-name-1,process-name-2)

must be established by the first message, which specifies a generic process name to cause the other MSG to create (or allocate) a receiving process.

The first Exchange operation, an Exchange OPEN ("EXOPEN"), does not send any data; however, we could have made Exchange more like MSG by combining the functions of EXOPEN and EXCH (the data-transfer primitive of the Exchange [9]) into a single call.

(4) Buffering

Buffering and concomitant flow control are fundamental to MSG. The Exchange, in contrast, deliberately avoids the flow-control issue by not providing any buffering.

(5) Generic Process Allocation and Creation

M*S*G functions as a process allocator, finding (or creating) a process to answer a generically-addressed message. The exact manner of doing this is not precisely defined in the MSG protocol description [6]. The options are as follows:

* M*S*G might allocate one from a pool of existing processes. These might be TSO jobs which it has previously logged on (or which it knows someone else has logged on). These might be separate jobs, or for a server like the File Package could be logical slots within a job which internally multiprograms.

* M*S*G might create a new process, e.g., by logging onto TSO as a pseudo-user or by submitting a batch job.

* M*S*G might simply queue the request until a process is available to service it.

* M*S*G might reject the request if no ReceiveGeneric request is pending from a local process.

The actual CCN rules are described in Section 4. Note also that the selection of a remote host to receive a generically-addressed message is controlled by tables within each MSG host, so that new generic names cannot be introduced dynamically.

(6) "Centralized Receiving"

Under MSG, send and receive operations work differently. MSG send operations specify the process name of the receiver, but receive operations do not specify the name of an acceptable sending process. When a particular receive does complete, therefore, the receiver must examine the process name of the sender (included with the message by the originating MSG) to determine the source of the message.

In short, an MSG process has only a single logical receive channel. If a process were to have more than one ReceiveSpecific operation pending, all of these operations would be equivalent. Therefore, there is no loss of generality in restricting a user process to one pending receive operation of each type (Generic, Specific, or Alarm); the CCN implementation of MSG makes this restriction, for simplicity.

In contrast, the Exchange was designed with complete symmetry between sender and receiver, and therefore has no comparable single-channel reception.

(7) Built-in Timer

Every MSG call for an asynchronous operation is specified to include a time-out value. A timing service built into an IPC mechanism is probably very useful; we rejected it in the design of the Exchange only because implementation under OS/MVT would have been difficult. Perhaps we gave up too easily; a privileged system task could periodically wake up, examine the Exchange queues for timed-out requests, and force them to complete abnormally.

(8) System Incarnation Number

Reliable operation of any interprocess communication system requires that a process not be able to accidentally reuse the handle for a communication path which is no longer open. Because the Exchange was designed to operate within a single CPU, it could rely upon an addition to the OS/MVT Supervisor to close an Exchange window if one task using it terminates or "abends" (terminates abnormally) without closing the window.

This approach will not work in a network; there is no great "abend-in-the-sky" to clean up when a single host crashes. Instead, MSG depends upon each host supplying an "incarnation number" for its MSG control program. Each time the local M*S*G is restarted, the incarnation number should be different, and the same number should not be reused "too often". When one MSG controller connects to another via the ARPANET, they begin with an initial handshake in which incarnation numbers are exchanged. The presumed current incarnation number is included in every transaction (as part of the process name), allowing the receiver to check it.

An equivalent scheme, based upon a globally-unique id, could have been used in the Exchange implementation, obviating the need for an OS/MVT Supervisor change.

1.3. MSG Implementation at CCN

We begin by enumerating explicitly the chosen objectives for the CCN implementation of MSC.

* Support TSO (Swapped) Processes

  The NSW processes which will use MSG will be the Foremen, interactive tools, and the File Package; all of these are expected to execute as TSO jobs. Thus, MSG must be able to create or allocate TSO jobs as MSG processes and to communicate with these jobs.

* Full Process-Callable Interface

  The process-callable interface to MSG should be as close as possible to that recommended in the MSG document [6]. This is achieved by providing a set of interface subroutines, i.e., an "access method", for user processes at CCN.

* Support PL/I Programs

  The MSG access method should be particularly suited for PL/I callers, because CCN is using PL/I as a higher-level system programming language for NSW.

* Responsiveness

  In order to be responsive to both local and remote processes, MSG should be implemented either at the operating system level (i.e., as an SVC) or else as a process (task) within a system job with high CPU priority. Writing and debugging complex programs at the SVC level is difficult, since these routines operate within the Supervisor protection sphere. Hence, the system job choice is preferable.

  In addition, efficient communication with swapped TSO jobs without "clogging the pipeline" to remote MSG's requires a pool of resident buffers for messages.

From these considerations, it was determined that: (1) MSG code should be divided between a central controller, M*S*G, and a local process access method called PL/MSG; (2) M*S*G should be implemented within a high-priority system task; and (3) for reasons of modularity and functionality, CCN's standard IPC mechanism the "Exchange" [9] should be used to communicate between M*S*G and PL/MSG instances. Fortunately, the Exchange supports communication between a resident process (e.g., M*S*G) and a swapped TSO process.

The design of the CCN Network Control Program (NCP) [10] assumes an NCP process per ARPANET connection; thus, the Network transmission for MSG must be handled by processes within the CCN NCP. In fact, the NCP itself runs as a high-priority task within a system job, so the NCP provides a natural environment for implementing M*S*G.

MSG must maintain a dynamic data base describing messages, alarms, connections, and processes. In a real-memory machine like the 360/91, this data base must be resident in main memory for responsiveness. This raises the problem of a location for this data base. The memory zone used by OS/MVT for system control blocks, the System Queue Area or "SQA", is in the Supervisor's protection domain (storage key zero). For reliability, we do not want any part of MSG to be privileged, so M*S*G could not use SQA for its data. Fortunately, the NCP region provides a suitable zone for allocating the MSG tables and queues.

Finally, the NCP uses a commutator (coroutine) control mechanism that runs as a single real task while simulating multiple processes ("pseudo-tasks" or "ptasks"). This mechanism simplifies the implementation of M*S*G, because it avoids most problems of mutual-exclusion from critical sections.

The objectives listed above controlled the design and implementation of MSG at CCN. However, MSG may have applications beyond NSW. In particular, MSG may be useful at CCN in two other ways:

* Support non-Arpanet communication to another host.

  In the future, CCN is likely to be operating large-scale CPU(s) in addition to the 360/91. If an additional machine were an IBM System 360/370, great flexibility would result from extending the Exchange to provide interprocess communication between the new CPU and the 360/91. In any case, a CPU-CPU IPC mechanism would be very useful and could be based on MSG. For example, the CCN NCP could be extended with additional I/O Driver processes to communicate with local CPU's as well as with the IMP.

* Support communication between two different TSO jobs

  MSG allows any two processes to communicate, regardless of whether they are in the same or different hosts. In particular, the CCN M*S*G handles intra-91 MSG messages efficiently, placing them directly on the receiver process queue without going through the IMP. Because MSG provides buffered communication, it will allow one swapped TSO job to talk to another.

2. PROCESS INTERFACE PACKAGE: PL/MSG

This section describes PL/MSG, a package of interface subroutines which a process executing on an IBM 360/370 can use to invoke the functions of the interprocess communication facility MSG. As such, this section supplements the primary NSW documentation on MSG [6] to provide the information required by a programmer writing an NSW process for an IBM 360/370.

The following discussion will be clarified by an understanding of the relationship between PL/MSG and MSG itself. The MSG protocol definition in NSW documentation [6] specifies a set of process-callable communication primitives. It is recommended that each MSG implementation provide a set of calls which are as nearly as possible identical to those described in the document. However, there may be differences from one implementation to another because of local system differences.

PL/MSG is the realization of this process-callable interface in the CCN implementation. PL/MSG is in fact a package of subroutines which cause MSG primitives to be executed; in IBM terminology, PL/MSG forms an "access method" for interprocess communication within the NSW. In addition, some PL/MSG calls perform non-primitive or support functions for MSG, not explicitly defined by the MSG protocol document [6].

PL/MSG is designed to be simple to call from processes written in PL/I, and in particular is compatible with object programs created by the IBM PL/I Optimizing Compiler [13]. However, most of the PL/MSG routines can be invoked from Assembler language or any higher-level language capable of supplying the necessary data structures. Furthermore, to minimize compiler dependencies, the PL/MSC routines do not depend upon PL/I data descriptors ("dope vectors") in their calls. This requires that the PL/I program declare the entries with the attribute:

```
OPTIONS (ASSEMBLER, INTER)
```

This will be done automatically if the %INCLUDE packets described later are used to declare the entry attributes. See the section entitled "CANNED PL/I DECLARATIONS."

PL/MSG opens and manages one or more Exchange windows to the central MSG control program M*S*G, which resides within the CCN Network Control Program. However, neither the Exchange protocol nor the internal function of M*S*G are important to the programmer writing a process using PL/MSG.

2.1. Basic PL/MSG Calls

In this section, we will describe the PL/MSG access method calls for other than direct connections. There are PL/MSG calls for MSG primitive operations (i.e., those described in [6]) and for some non-primitive operations. The actual functions of MSG primitives are described in the NSW MSG documentation [6], with which the reader is assumed to be familiar.

The PL/MSG call descriptions in this section are written in terms of invocation from PL/I programs; however, these calls can be made from any language that can supply the necessary data structures. The actual data types for the parameters to the calls are defined later, in the section "PL/MSG Data Structures". The values returned in event signals are described in appendix A.

A PL/MSG call is non-blocking; that is, it returns control to the caller immediately whether or not the operation has completed. When the operation does complete, the caller is signaled through an "event signal" variable passed as part of each call. The same event signal variable is used to return an indication of the success/failure of the operation. In general, the caller should not examine any of the results of the operation until its successful completion has been signaled. Furthermore, in the case of a send operation, the caller should not change the "message area" containing the message to be sent until the send operation is signaled complete.

2.1.1. Materializing a Process ("TerminationSignal")

```
CALL MSGMP  (process name,     /* Set and returned */
             process handle,   /* PL/MSG returns   */
             event signal);    /* PL/MSG signals   */
```

This non-primitive operation "materializes" the current process, i.e., "introduces" it to MSG, assigning a specific process name and providing an event signal which M*S*G can use to request graceful termination.

A process must issue this materialization call before it can issue other PL/MSG calls. For example, when M*S*G creates a new process in response to a SendGeneric from a remote host, the new process should issue this materialization call immediately upon starting, before issuing a matching ReceiveGeneric call. However, a process that was not started automatically by M*S*G can still materialize. This simplifies debugging of processes, by allowing the programmer to start a process under direct control of a TSO terminal rather than indirectly through the Front End, Works Manager, and Foreman. M*S*G will not let an arbitrary process materialize, however; the jobname of the process must match an entry in a table of valid process jobnames in an MSG configuration module, as described later.

"Event signal" corresponds functionally to the MSG primitive "TerminationSignal", and in fact replaces that primitive in this implementation.

"Process name" should be the chosen generic name, with all fields except "generic name" zero. PL/MSG returns values for the other fields, yielding the specific process name. The operation also returns a handle to the process in "process handle", but the process has to save and use this handle only if it needs to materialize multiple MSG processes within the same load module instance; see the section entitled "MATERIALIZING MULTIPLE PROCESSES."

In the future, it is planned to define a specific form of "process name" which will allow an aborted process to restart itself and reestablish interrupted communication with M*S*G. Definition of this case is deferred.

2.1.2. "ReceiveGenericMessage"

```
CALL MSGRGM  (message area,      /* Set and returned */
              process name,      /* PL/MSG returns   */
              event signal,      /* PL/MSG signals   */
              time-out interval);/* Caller  sets     */
```

The local process calling PL/MSG must give a maximum value to the length field of "message area". PL/MSG will set a true length when it fills in the message data, and will also return a specific process name.

2.1.3. "ReceiveSpecificMessage"

```
CALL MSGRSM  (message area,      /* Set and returned */
              process name,      /* PL/MSG returns   */
              event signal,      /* PL/MSG signals   */
              time-out interval, /* Caller sets      */
              special handling); /* PL/MSG returns   */
```

The caller must give a maximum value to the length field of "message area". PL/MSG will set a true length when it fills in the message data, and will also return a specific process name and values for "special handling".

2.1.4. ReceiveAlarm ("EnableAlarm")

MSG documentation refers to this primitive as "EnableAlarm". We will call it "ReceiveAlarm" in CCN documentation, because the original name has been found to cause confusion.

```
CALL MSGRA( alarm code,          /* PL/MSG returns   */
            process name,        /* PL/MSG returns   */
            event signal,        /* PL/MSG signals   */
            time-out interval); /* Caller sets      */
```

PL/MSG will fill in "alarm code" and all parts of "process name".

2.1.5. "SendGenericMessage"

```
CALL MSGSGM  (message area,      /* Caller sets      */
              process name,      /* Caller sets      */
              event signal,      /* PL/MSG signals   */
              time-out interval, /* Caller sets      */
              wait enable);      /* Caller sets      */
```

The caller must give values to "message area", "process name", and "wait enable". "Process name" must be a generic name, with zeros in all fields except "generic_name", and optionally, "host_number".

2.1.6. "SendSpecificMessage"

```
CALL MSGSSM  (message area,      /* Caller sets      */
              process name,      /* Caller sets      */
              event signal,      /* PL/MSG signals   */
              time-out interval, /* Caller sets      */
              special handling); /* Caller sets      */
```

The caller must give values to "message area", "process name", and "special handling". "Process name" must be a specific name with all fields specified by the caller.

2.1.7. "SendAlarm"

```
CALL MSGSA   (alarm code,        /* Caller sets      */
              process name,      /* Caller sets      */
              event signal,      /* PL/MSG signals   */
              time-out interval); /* Caller sets     */
```

The caller must give values to "alarm code" and to all parts of the specific "process name".

2.1.8. "Rescind"

```
CALL MSGRSND (event signal); /* Caller sets */
```

In this special case, the "event signal" variable is not associated with the completion of the Rescind primitive (which does not itself create a pending event), but rather identifies the pending event to be rescinded. The "event signal" variable will be signaled with a disposition code to indicate that the event with which it was originally associated was aborted by a Rescind. However, if it is found to be already signaled complete, this call becomes a no-operation. Notice that the ADDRESS of "event signal" is actually used to locate the original event, so passing a copy will not work.

2.1.9. ArmAlarms ("AcceptAlarms")

MSG documentation refers to this primitive as "AcceptAlarms". We will call it "ArmAlarms" in CCN documentation, because the original name has been found to cause confusion.

```
CALL MSGAA (alarm arm); /* Caller sets */
```

where "alarm arm" is '1'B to arm the process for alarms, and '0'B to disarm.

2.1.10. "Resync"

```
CALL MSGRSNC (process name); /* Caller sets */
```

where "process name" must be a specific name with all fields specified by the caller.

2.1.11. "Stopme"

```
CALL MSGSTOP;
```

In the CCN implementation, this call dematerializes the current MSG process. It does not affect program or task status in any other way. On return, the calling program is free to materialize another process, terminate, or continue execution dealing with non-MSG work.

2.1.12. Waiting for Events

```
CALL MSGWAIT (event signal, ...);
```

This routine does not correspond to an MSG primitive, but is provided as a programming convenience. Its parameter list consists of any number of event signal variables. Control does not return until at least one of the event signals is posted.

2.1.13. Set User Post Subroutine

```
CALL MSGSETP ( post subroutine,
               bit mask /* optional */) ;
```

This non-primitive PL/MSG routine causes a change in the method of signaling ("posting") event signal variables for the subset of PL/MSG routines defined by "bit mask". It basically provides an elementary software interrupt mechanism for these event signals.

"Post subroutine" can be a PL/I POINTER or ENTRY variable which contains the entry point address of a closed Assembly-language subroutine. For all subsequent posting of user-provided event signal variables for the specified routines, PL/MSG will call this subroutine instead of posting the event variable. If "post subroutine" is a NULL POINTER, PL/MSG will revert to standard posting of these event variables.

Each PL/MSG routine is assigned a unique index called the "primitive number"; see the discussion of event signal variables in Section 2.3.3. If substr("bit mask", P, 1) = '1'B, then MSGSETP will change the posting method for calls to the routine with primitive number P. If bit P is '0'B, then the posting method for P will be unchanged by this call. The "bit mask" argument may be omitted, in which case the default '111...1'B will be used, i.e., all PL/MSG routines will be affected.

A post subroutine must be written in Assembler language and be capable of operating as a software interrupt routine (i.e., on an IRB). The registers must be used as follows:

R13: the address of a usable save area;
R14: the return address;
R15: the entry address of the routine;
R0 : the post code;
R1 : the ECB address;
R2-R12: undefined-- must be restored if altered.

2.1.14. Host Type

```
CALL  MSGHTYP (  host number,  /* FIXED BIN(15),  */
                               /*   caller sets   */
                 host name,    /* CHAR(*) VARYING,*/
                               /*   PL/MSG returns*/
                 host family ) /* CHAR(*) VARYING,*/
                               /*   PL/MSG returns*/
```

This non-primitive routine returns information about a particular ARPANET host. A process is encouraged to use this function to look up host information in the MSG configuration tables, rather than to have its own host table.

The caller must set "host number", which is the same as the corresponding subfield of a PL/MSG Process Name structure (see definition in Section 2.3.1 below).

"Host name" and "host family" are character strings returned by PL/MSG. Conventionally, "CHAR(32) VARYING" strings are used. These strings can be used in human-readable messages, or can be compared to determine family relationships.

2.1.15. Editing Process Names

```
edited string= PNAMOUT( addr(process name) );
```

This function converts a standard PL/MSG process name structure (defined below in Section 2.3.1) into a printable character string for human consumption.

"Edited string" may be any VARYING character string variable, conventionally of length 128 maximum.

2.2. Direct Connections

There is a separate set of PL/MSG calls for manipulating MSG direct connections. Unlike the calls listed in the preceding section, the direct connection subroutines are written in PL/I and can only be called by PL/I-compiled programs. No OPTIONS attribute should be declared for these entries. This difference will be transparent to the PL/I programmer using the %INCLUDE packets (described in the subsection entitled "CANNED PL/I DECLARATIONS") to declare the attributes.

2.2.1. Opening a Direct Connection ("OpenConn")

The particular CCN implementation of OpenConn and CloseConn makes them technically non-primitive operations. However, the only effect of this on the PL/MSG caller is that an OpenConn call can be rescinded only by calling CloseConn, not by calling Rescind.

```
CALL MSGOC (connection request, /* Caller sets      */
            process name,       /* Caller sets      */
            time-out interval,  /* Caller sets      */
            connection handle,  /* PL/MSG returns   */
            open event signal,  /* PL/MSG signals   */
            close event signal);/* PL/MSG signals   */
```

The caller is responsible for ensuring that the "connection identifier" subfield of "connection request" is known to the remote process, which must specify the same value in its matching OpenConn request. On return from this call, even before the event is signaled complete, PL/MSG will have assigned a value to "connection handle". The caller should save this handle in order to reference the connection in subsequent PL/MSG calls.

The "open event signal" variable will be posted when the connection has opened successfully. The "close event signal" will be posted whenever the connection is closed, for one of the following reasons (as indicated by the "disposition" field of the event signal):

1) The connection failed to open, due to a reason known to M*S*G.

2) A CloseConn was issued by this process.

3) A CloseConn was issued by the remote process.

4) An unrecoverable failure occurred within the communications machinery between the two processes.

Hence, after calling MSGOC the process should wait on both the open event and the close event in order to be certain that the connection opened successfully. In any case, if the close event is signaled before this process has issued a CloseConn, the caller should: (1) dispose of all pending GET's and PUT's for the window (all will have completed in one way or another, and GET's may have provided the caller with good data); and (2) issue CloseConn.

2.2.2. Closing a Direct Connection ("CloseConn")

```
CALL MSGCC (connection handle); /* Caller sets */
```

This primitive requests that MSG close the direct connection which is identified by "connection handle", the value returned by OpenConn. In the CCN implementation, this primitive rescinds any pending events (open, GET, or PUT) for this connection before requesting the close.

Completion of the close is signaled through the "close event signal" variable passed to PL/MSG by the OpenConn call. If that event signal (ECB) was posted before this call, this call will complete immediately (though the call should still be made). The ECB will not be cleared or posted again, so the caller can still wait on it if convenient.

In order to avoid lost data, two processes sharing a connection may synchronize their use of CloseConn by means of the "higher-level" protocols used for communication on the connection. However, PL/MSG provides recommended sequences for avoiding lost data, minimizing dependence upon higher-level protocols. These sequences are as follows:

* Case 1 -- abort: The caller can use MSGCC to close any type of connection at any time, if the intention is to abort the associated activities.

* Case 2 -- responding to a close: Any time a connection's close event signal is posted, implying that the other process has initiated CloseConn, the caller should clean up and issue CloseConn as soon as practical. This is the recommended technique in the case of binary simplex receive connections.

* Case 3 -- binary simplex send: CloseConn can be issued at any time without loss of data.

* Case 4 -- binary full-duplex: To initiate CloseConn, use MSGEOD to "close" the send half of the connection. This has the effect of draining the local send buffers and posting the other process' close event signal. That process is then expected to close the connection, draining the local input buffers and posting the close event signal. The caller is then expected to call MSGCC.

* Case 5 -- TELNET types: These conversational connections require some higher-level-protocol synchronization. The only general rule is that the User TELNET side should normally be the aggressor.

2.2.3. Preparing for CloseConn

```
CALL MSGEOD (connection handle); /* Caller sets   */
```

MSGEOD is a preliminary to MSGCC. It can be issued against any connection, but it only has a specific function in the case of binary full-duplex connections; it indicates that the caller intends to issue no more PUTs for the connection. However, the caller may still issue GETs. See the recommended usage of MSGEOD in Case 4 under "Closing a Direct Connection", above.

2.2.4. Getting Data from a Direct Connection

```
CALL MSGGET (connection handle,/* Caller sets      */
             message area,     /* Set and returned */
             event signal);    /* PL/MSG signals   */
```

The caller must have set a maximum length in "message area". PL/MSG will store data from the connection identified by "connection handle" in "message area", adjusting the length of "message area" to the actual data received. The number of 8-bit data bytes returned from a single MSGGET depends on the connection type:

* Case 1 -- TELNET types: MSGGET returns a data string whose length is the smallest of: (1) the initial length of the character string "message area", (2) the number of bytes received from the remote process, or (3) the smallest substring ending in either a "Newline" or a "Goahead" character. It will only wait if no data has arrived from the remote process. In an extreme case, the caller may receive input one character at a time.

The representations for "Newline" and "Goahead" depend upon the options set for the CCN TELNET protocol handlers. The mechanism for setting TELNET options is independent of MSG and is documented elsewhere [15].

* Case 2 -- Binary types: MSGGET will not complete until the message area has been filled, or until the local system learns that the remote process has initiated CloseConn.

MSGGET always returns an integral number of 8-bit bytes. If the connection byte size is not a multiple of 8, it is the responsibility of the caller of MSGGET to interpret the buffer as a bit string and locate logical byte boundaries. Excess bits at the end of one such string must be saved and concatenated ahead of the next string received. Padding bits following the last complete logical byte may be discarded.

PL/MSG gives the receiving process the message length as a count of 8-bit bytes. This can be unambiguously converted to a count of connection bytes only if the latter are 8 bits or larger. Hence connection byte sizes of less than 8 bits are useful only if the receiver can learn the message length through some other means.

2.2.5. Sending Data over a Direct Connection

```
CALL MSGPUT (connection handle /* Caller sets      */
             message area,     /* Caller sets      */
             event signal);    /* PL/MSG signals   */
```

The caller must provide output data in "message area", and must not alter it until the event signal is posted.

MSGPUT always accepts an integral number of 8-bit bytes. If the connection byte size is not a multiple of 8, it is the responsibility of the caller of MSGPUT to concatenate logical bytes into a free bit string for MSGPUT. Bits that overreach the last 8-bit byte boundary must be saved and concatenated ahead of the next string to be transmitted. The very last logical byte must be padded by the caller to the nearest multiple of 8 bits.

Note that the message length is specified to PL/MSG in units of 8-bit bytes; this can be unambiguously converted to logical bytes only if the latter are 8 bits or larger.

2.2.6. Receiving TELNET Attentions

```
CALL  MSGRAT ( connection handle, /* Caller sets  */
               event signal )     /* PL/MSG sets  */
```

This call enables the process to receive "Attention" signals from a Server TELNET direct connection. This call is ignored for other connection types.

An Attention will cause "event signal" to be posted and the Attention signal to be "disabled" (i.e., a later Attention will be queued until MSGPAT is called again). A maximum of one Attention will be queued while Attentions are disabled; further Attentions will be discarded. Note that a Server TELNET connection cannot be disarmed for Attentions; the fact that an Attention occurred is always remembered. Closing the connection, however, will discard pending Attentions.

2.2.7. Sending TELNET Attentions

```
CALL  MSGSAT ( connection handle, /* Caller sets  */
               event signal )     /* PL/MSG sets */
```

This call is ignored for all connection types except User TELNET. It transmits an Attention signal through the connection.

2.3. PL/MSG Data Structures

This section describes the data types for PL/MSG parameters used in the calls listed above. Each is described in terms of a PL/I data declaration, and its values are discussed in general terms. Any details on the use of a data structure in the context of a particular PL/MSG call were given in the preceding section.

2.3.1. PROCESS NAMES

A "process name" is a structure of the form:

```
DCL 1 process_name,
      2 host_number      FIXED BIN(15),
      2 host_incarnation FIXED BIN(15),
      2 process_instance FIXED BIN(15),
      2 generic_name     CHAR(127) VAR;
```

For a generic name, only "generic_name" need have a non-null value. "Host_number" may have the value assigned to one of the NSW host computers or may be zero to indicate that MSG may choose any host supporting "generic_name". The other two integers must be zero for a generic name.

For a specific name, all three integers must be non-zero. The calling program will have obtained the complete (specific) name from M*S*G or from a message from another process.

When receiving a process name from PL/MSG, the local process must declare "generic_name" to be CHAR(127) VARYING. When it is being passed to PL/MSG, this string need be declared only large enough to accommodate the actual value, but must always be VARYING.

2.3.2. PROCESS HANDLES

"Process handle" is a variable of the form:

```
DCL process_handle POINTER;
```

It is generated by PL/MSG at process materialization time and may be used by the calling program as a process handle. However, the caller does not need to retain this value unless it is materializing multiple concurrent processes within the same program. See the section entitled "MATERIALIZING MULTIPLE PROCESSES" for more information.

2.3.3. EVENT SIGNALS

A PL/MSG "event signal" is a variable of the form:

```
DCL event_signal FIXED BIN(31);
```

It is used as an OS/360 Event Control Block (ECB) whose contents may be represented by the following based structure:

```
DCL 1 ecb BASED,
      2 (waitbit, postbit, dummy(6)) BIT(1),
      2 primitive_no  BIT(8),
      2 disposition   FIXED BIN(15);
```

A PL/MSG event signal is used both to signal the completion of an MSG pending event and to determine whether the event completed normally. Thus it serves the functions of both <signal> and <disp> as documented in the NSW MSG specifications.

The caller never needs to set or clear an "event signal" variable. However, the caller does need a method of "waiting on the ECB", i.e., blocking his process until the event completion is signaled by PL/MSG. The caller also needs to be able to read the "disposition" value that PL/MSG returns in the event signal variable.

Notice that a PL/MSG "event signal" variable (ECB) is different from a PL/I event variable, although the latter does contain or point to an ECB. Therefore, an "event signal" cannot be used directly in a PL/I WAIT statement, but the caller can use any of several other methods. One alternative is the MSGWAIT subroutine described earlier. The caller may also test for the completion of a pending event with a statement like:

```
IF ADDR(event_signal)->postbit
   THEN /* event-complete processing */
```

In any case, do not assume that a pending event is complete, and do not alter its supplied parameters or use its returned values, until the caller has detected PL/MSG's completion signal.

The "disposition" portion of the event signal is set simultaneously with POSTBIT. The caller can test it with a statement of the form:

```
IF ADDR(event_signal)->disposition ^= SUCCESSFUL
   THEN /* abnormal completion processing */
```

The values of "disposition" are defined uniformly across all PL/MSG calls. The PL/I programmer should not be concerned with the numeric values of "disposition", but should associate names with them through %INCLUDE packet MSGDISP. See Appendix A for the names currently defined in that packet and their meanings.

The portion of the event signal called "primitive_no" in the above declaration is set by PL/MSG to the primitive number for the routine whose pending event is being signalled. Most PL/MSG callers will not need to examine this; however, it will be useful for debugging. Its values are indexes, i.e., fixed binary numbers. The PL/I programmer can associate routine names with these values through the %INCLUDE packet MSGPNO.

2.3.4. MESSAGE AREAS

A PL/MSG "message area" is a character variable of the form:

```
DCL message_area CHAR(nn) VAR;
```

where "nn" is chosen to be compatible with the calling program's operation. However, it is useful to redeclare it in the following form:

```
DCL message_length BASED FIXED BIN(15) UNALIGNED;
```

When the local process is passing a message to PL/MSG, the normal act of assigning a value to the string will set its length field properly. When requesting a message from PL/MSG, the calling program must specifically set the length field to the maximum length that it is able to accept -- normally "nn". The most efficient way to do this is:

```
ADDR(message_area)->message_length = nn;
```

2.3.5. TIME-OUT INTERVALS

A PL/MSG "time-out interval" parameter is a variable of the form:

```
DCL time-out FIXED BIN(31);
```

It is set by the caller of a PL/MSG entry that creates an MSG pending event, to specify a time-out interval, in hundredths of real seconds, after which the request will be aborted and signaled. A value of zero represents an infinite interval.

2.3.6. SPECIAL HANDLING CODES

A PL/MSG "special handling" parameter is a structure of the form:

```
DCL 1 special_handling,
      2 (sequenced, stream_marker) BIT (1) UNALIGNED;
```

If a bit is set, it requests or reports the use of the corresponding MSG special handling feature for the associated message.

2.3.7. ALARM CODES

A PL/MSG "alarm code" parameter is a variable of the form:

```
DCL alarm_code FIXED BIN(15);
```

Alarm codes take integer values in the range -32768 through 32767.

2.3.8. WAIT ENABLES

A PL/MSG "wait enable" parameter is a variable of the form:

```
DCL wait_enable BIT (1) ALIGNED;
```

A value of '1'B indicates that a SendGenericMessage is to cause an immediate error response (rather than being held pending) if a matching ReceiveGenericMessage is not already pending.

2.3.9. BIT MASK

A PL/MSG "bit mask" parameter is a bit string of the form:

```
DCL bit_mask BIT(*) VARYING;
```

If the bit SUBSTR(bit_mask,P,1) is '1'B, the routine with primitive number P is to be included in the set.

2.3.10. CONNECTION REQUESTS

A PL/MSG "connection request" is a structure of the form:

```
DCL 1 connection_request,
      2 type          CHAR (4),
      2 byte_size     FIXED BIN(15),
      2 connection_id FIXED BIN(15),
      2 queue_depth   FIXED BIN(15);
```

where "type" is one of the strings:

* 'STEL' for a server TELNET connection;

* 'UTEL' for a user TELNET connection;

* 'FULL' for a binary full duplex connection;

* 'SEND' for the sending end of a binary simplex connection;

* 'RECV' for the receiving end of a binary simplex connection.

"Byte_size" may be between 1 and 255, but will usually be 8 or 32. Byte sizes of less than 8 bits present special problems to the programmer, and are thus discouraged. This value is ignored for TELNET connections, which always have byte size 8.

"Connection_id" is a caller-supplied connection number known to the remote process. The remote process must specify the same value in order to connect the two ends of the connection properly.

"Queue_depth" states the maximum number of MSGGET and MSGPUT events that can be pending on the connection at any given time. This must be a small positive integer.

2.3.11. CONNECTION HANDLES

A PL/MSG "connection handle" parameter is a variable of the form:

```
DCL connection_handle POINTER;
```

The value of this variable is returned from OpenConn and is used in subsequent operations to identify the direct connection.

2.3.12. ALARM ARMS

A PL/MSG "alarm arm" parameter is a variable of the form:

```
DCL alarm_arm BIT (1) ALIGNED;
```

A value of '1'B indicates that alarms can be accepted, '0'B indicates that they should be rejected.

2.4. Programming with PL/MSG

2.4.1. PENDING-EVENT SET

Most PL/MSG operations are asynchronous, completing at some indeterminate time after control is returned to the caller. Each of these operations, when issued, becomes a "pending event" that is a member of the "pending-event set" for the process.

There are some limitations upon the size and composition of the pending event set, i.e., limitations on the number of operations of each type that can be pending at the same time.

* There may be a maximum of one pending operation for each of the Receive types:

  ReceiveGenericMessage
  ReceiveSpecificMessage
  ReceiveAlarm

  at the same time. PL/MSG will detect a violation of this rule and refuse a second operation with an error return code "ALREADY_PENDING".

* The number of Send events is effectively unlimited, except by the following:

* The total number of pending events of all types may not exceed 25.

2.4.2. MATERIALIZING MULTIPLE PROCESSES

In order to materialize more than one MSG process, the local program must declare and manage a STATIC EXTERNAL variable with the name PROCESS and the format of a "process handle". This variable is a handle to PL/MSG's tables for a single NSW process. It is set by MSGMP to the value returned by that call in its "process handle" parameter. So long as this value is unchanged, subsequent PL/MSG calls will be on behalf of the last-materialized process.

In order to function as several MSG processes concurrently, a program must maintain its own set of process-control blocks, with each containing the associated process handle. This value should be restored to PROCESS whenever the program changes its process identity.

Each materialized process will remain known to MSG until a Stopme primitive is executed with the corresponding value in PROCESS, or until the task that was in control at process materialization terminates.

Notice that, due to PL/MSG's use of static storage to communicate the process handle, a load module containing PL/MSG cannot be reentrant. This means that such a program MUST do its own "process management". The caller cannot rely on external multi-tasking support, even if the program is otherwise reentrant.

2.4.3. BUFFER LENGTH BOUNDS

All PL/MSG buffers are constrained by their PL/I-compatible representation to have lengths in the range 0-32767 bytes. Null messages are allowed, but PL/MSG will return an error if the caller passes a zero-length buffer to "ReceiveSpecificMessage" or "ReceiveGenericMessage". In the case of direct connection buffers, any lengths in the range 1-32767 can be handled with reasonable efficiency. In the case of buffers for MSG messages, the upper bound on tolerable buffer length depends on network activity, and cannot be stated precisely. In general, message buffers of more than 1000 bytes (8000 bits) are considered unreasonable.

2.4.4. CODE TRANSLATION

By ARPANET convention, character transmission uses the ASCII-68 character set. Since PL/MSG is not sensitive to the forms of messages that it handles, character parts will be in ASCII in the PL/MSG caller's storage. It is up to the caller to provide appropriate translations in both directions.

The character portions of process names that are used explicitly in PL/MSG calls, i.e., not embedded in MSG messages, are always expressed in EBCDIC in the caller's storage.

Data transmitted over a direct connection is encoded by mutual agreement of the two owners of the connection. TELNET direct connections are exceptions; unless the local process has arranged other options with the CCN TELNET protocol handlers, these routines will translate TELNET data to/from EBCDIC in local storage.

2.4.5. INSERTING PL/MSG INTO A MODULE

PL/MSG will be included in the load module of any program which references its entry points, provided the appropriate subroutine libraries are available to the Linkage Editor. However, the module entry point must be

altered to support MSGBUG, and this is not automatic. To ensure a correct entry point, add these statements to the Linkage Editor input (this example is only for a PL/I program to be executed as a TSO command):

```
INCLUDE SYSLIB(MSGSTART)
ENTRY MSGSTART
```

In addition to the modules bound at linkage-edit time, the module with aliases MSGSTAX, MSGCMD, and MSGEDIT must be available for dynamic loading during program execution. This means that the library containing it must be a part of the task library structure, for example a library defined via a STEPLIB card in the LOGON procedure. See Section 4 and Figure 7.

2.4.6. CANNED PL/I DECLARATIONS

For the convenience of the PL/I programmer, certain declarations will be stored in a public library. These can be invoked through the PL/I %INCLUDE statement. Because unneeded entry point declarations can result in the inclusion of extra subroutines in the final load module, each PL/MSG entry has been stored under its own name. The caller should select the declarations for the procedures to be used, for example:

```
%INCLUDE MSGRGM, MSGSSM, MSGDISP;
```

The available %INCLUDE names are listed below:

```
MSGAA
MSGCC
MSGDISP (event signal values -- disposition)
MSGEOD
MSGGET
MSGMP
MSGOC
MSGPNO (event signal values -- primitive number)
MSGPUT
MSGRA
MSGRGM
MSGRSM
MSGRSNC
MSGRSND
MSGSA
MSGSGM
MSGSSM
MSGSTOP
MSGWAIT
```

3. MSG DEBUGGING INTERFACE: MSGBUG

PL/MSG interfaces with a user process via the subroutine calls described earlier in this document, and with M*S*G within the CCN NCP via a protocol that is strictly internal. For MSG process debugging, a third interface has been added: MSGBUG is a human interface directly to the TSO terminal controlling the session.

MSGBUG is present in all PLIX programs that use PL/MSG, but it cannot be activated for a process running in batch mode. Furthermore, MSGBUG is only marginally useful for TSO sessions unless the remote user has access (e.g., via a TELNET direct connection) to the TSO virtual terminal controlling the session. If the session is under the control of MSG's process-creation mechanism, then at present the user can access the virtual terminal only by opening a direct TELNET connection of type 'TCAM'; this option has been provided in the CCN MSG for the purpose of encapsulation [8]. Otherwise, MSGBUG can provide only passive monitoring of the MSG traffic to the process (Note: the passive monitoring feature is not operational at the time of this writing.)

On the other hand, an NSW process being debugged under TSO may be invoked and executed by a human-driven terminal as well as by MSG, making MSGBUG a useful interactive dubugging tool (in conjunction with the TSO TEST facility.)

MSGBUG includes a command interpreter, as described in the next subsection. To reduce confusion with other command levels, the MSGBUG commands are referred to as "orders".

3.1. MSGBUG Command Language

The set of MSGBUG orders is intended to be easily expandable. The present set is minimal and is known to be insufficient, particularly in the areas of direct-connection support and actual status modification. MSGBUG will usually operate under the TSO TEST processor [14], so TEST functions have not been duplicated except where specifically advantageous.

The present set includes status-switching orders, which turn MSGBUG functions on or off, and information request orders, which report recent history and/or status of MSG operations. The actual orders are as follows:

* HELP

  prints a list of valid order names (but does not explain them).

* CONTINUE

  exits from the MSGBUG command level and resumes execution of the program using MSG.

* STATUS

  prints messages defining the current status of the MSG process, including details about each pending event.

* RECENT[(n)]

  prints the newest "n" entries in the MSG event trace table. If "(n)" is omitted, the entire table is printed. The order of printing is always newest to oldest. RECENT requires that the tracing function be active.

* LOG[(logid)]

  turns on the logging function. This causes real-time printing of trace table entries; that is, they are listed at the time they are to be entered into the trace table. The tracing and logging functions are entirely independent, however, and either can be used without the other.

  "Logid" is an optional character string which will eventually be used to specify that logging is to be directed to a TSO terminal other than the one issuing the order. It is currently accepted but ignored.

* NOLOG

  turns off the logging function.

* TRACE[(traceno)]

  turns on the MSG tracing function and allocates a circular trace table that is "traceno" MSG events in circumference. If "traceno" is omitted, the last-specified value is used, and if none has ever been specified, a default of 20 is used. If "traceno" is specified, it is stored for future use.

  If TRACE is issued while the tracing function is active, NOTRACE is simulated first. This can be used to change the size of the trace table, but only at the expense of losing its current contents.

* NOTRACE

turns off the MSG tracing function and releases any allocated trace table storage. It does not alter the stored value of "traceno".

* VERBOSE

causes all subsequent output from RECENT, STATUS, or the logging function to use unabbreviated annotations that are largely self-explanatory. This is the default mode of operation.

* TERSE

causes all subsequent output from RECENT, STATUS, or the logging function to use highly-abbreviated annotations that are geared to the seasoned MSGBUG user. Use of TERSE can halve the volume of MSGBUG terminal output.

* DCTRACE[(dctraceno)]

turns on the direct-connection trace function and sets the size of each connection's circular trace table to "dctraceno". If "dctraceno" is omitted, the last-specified value is used, and if none has ever been specified, a default of 20 is used. If "dctraceno" is specified, it is stored for future reuse.

The direct-connection trace function is fixed for each direct connection at the time the corresponding OpenConn primitive is issued, and remains fixed for that connection until it is closed. DCTRACE does not affect tracing operations for currently open direct connections. If DCTRACE is issued while the direct-connection tracing function is already active, only the value of "dctraceno" is affected. If this is omitted, the order is a no-operation.

* NODCTRACE

turns off the DCTRACE function for direct connections opened subsequently. It does not alter the stored value of "dctraceno". It does not affect tracing operations for currently open direct connections.

* SNAP

invokes the OS/MVT SNAP service in the task using MSG(BUG). Upon successful completion of this order, a data set will have been created, filled with dump data, and enqueued for immediate printing. The routing of the printed output is determined by the path established for the TSO userid under which this session is logged on.

3.2. Entering MSGBUG Orders

MSGBUG accepts orders from, and prints replies to, the controlling TSO terminal. Orders are entered in groups of individual order names separated by blanks or commas, and terminated by a semicolon, carriage return, or newline. To be precise, they follow the syntax of keyword operands of a TSO command, as defined in [14]. Orders can be entered in two situations:

1) As operands of the TSO command that invokes the program that will materialize the process. Generally these will be status-switching orders to turn on MSGBUG functions. (To keep production processes efficient, all MSGBUG history functions are turned off by default.) Information requests are legal, but they have no useful information to report at this point.

   Assuming that appropriate task libraries are already established, the format of a TSO command to invoke a PL/I program using PL/MSG is:

   <program> ∅ <orders> ; <options> / <parameters>

   where:

   * "program" is the TSO command name, which is also the name of the load module to be executed.

   * "orders" are the initial MSGBUG orders.

   * "options" is a list of the execution-time options for the PL/I environment.

   * "parameters" is the parameter string for the PL/I MAIN procedure.

   For example:

   ```
   FILEPKG LOG,TRACE(50);ISA(4K)/GENNAME='FP'
   ```

2) As a stand-alone string, in response to the MSGBUG mode message. You attain the MSGBUG command level by signaling "attention" to TSO, in a manner appropriate to your particular real or virtual terminal. For simplicity of implementation, MSGBUG uses some slightly non-standard conventions on this command level, as follows:

   * "Attention" does not function as a line-delete only character. If you signal attention at any time on the MSGBUG command level, you will be transferred to the next higher command level (usually TEST or

READY), and anything else typed on the current line will be discarded.

* On return from a higher command level to MSGBUG, the outstanding MSGBUG mode message is not repeated.

* If MSGBUG is entered while the process being tested is waiting for terminal input, returning to the original command level may satisfy that wait with a null input string.

* Remember that any TSO control level escape via "attention" always purges any enqueued terminal output buffers.

3.3. MSGBUG Defaults and Priority

Until an explicit MSGBUG order is issued, the default state will be:

```
VERBOSE,NOLOG,NOTRACE,NODCTRACE
```

If an explicit value is never given for the "traceno" or the "dctraceno" operands of TRACE or DCTRACE, a default of 20 is used.

Any number of MSGBUG orders can be entered on a single command line. The order in which they are processed follows two rules:

* For conflicting parameters (e.g., LOG/NOLOG in the list below), only the last one entered is recognized.

* For non-conflicting parameters, regardless of the order of entry, the processing order is:

```
HELP
TERSE/VERBOSE
LOG/NOLOG
DCTRACE/NODCTRACE
STATUS
RECENT
SNAP
TRACE/NOTRACE
CONTINUE
```

4. MSG OPERATION AND CONFIGURATION

Section 2 of this report and [6] describe the syntax and semantics of the process-callable interface to the CCN implementation of MSG. We will now define what we may call the "pragmatics" of the calls, i.e., the relationships among both local and remote MSG users. In particular, we must describe the algorithms which the MSG control program M*S*G uses for flow control, process creation, and certain aspects of inter-MSG communication.

4.1. Host Incarnation Numbers

Whenever it is restarted, M*S*G generates a "unique" incarnation number by reading the system clock. Specifically, it uses the time since midnight, expressed as an integer in units of 2**16 timer units, which is approximately 1.707 seconds. The resulting incarnation number should be random, with a negligible probability of repeating within the longest lifetime of NSW.

When MSG's on two hosts open an ARPANET connection to each other, they exchange current incarnation numbers. CCN's MSG records the incarnation numbers of the remote MSG's, but it does not use this information. The local process which receives a message or alarm is expected to check the incarnation number as part of the sender's process name.

4.2. Generic Names

The MSG protocol divides processes into "generic classes" of equivalent function. Each generic class is assigned a "generic name", a character string of up to 127 characters. However, for efficiency, MSG also allows the optional use of 8-bit nicknames or "generic codes" for commonly-used generic classes.

The CCN MSG uses the following rules for generic names:

1) Generic codes are completely hidden from a local process, which can use only the full generic names in its PL/MSG calls and which will receive only full generic names from PL/MSG.

2) M*S*G will accept either a full generic name or a generic code in any inter-MSG protocol item that it receives from a remote MSG.

3) When sending a message to a remote MSG, M*S*G will encode the full name received from PL/MSG into a generic code, if such a code is defined in the appropriate configuration table.

4) When starting a local process in response to receiving a SendGeneric message, M*S*G must generally map the generic name into a load module or cataloged procedure name of 8 characters or less. In the current implementation, this mapping is not general; in fact, the first 8 characters of the full generic name are used. Therefore, the first 8 characters of names for different generic classes should be unique -- unless the remote MSG can guarantee to send generic codes rather than names.

4.3. Flow Control

The MSG protocol provides flow control by use of the inter-MSG data items MESS-HOLD and XMIT. The CCN implementation can hold messages in response to MESS-HOLD requests from remote MSG's and will later send them upon receiving corresponding XMIT items.

Since the IBM 360/91 does not have a virtual memory, CCN's M*S*G must allocate buffers from a limited zone of real storage, i.e., a subpool within the NCP region. Although processes using MSG are expected to be normally "well-behaved", exchanging relatively small numbers of small messages, M*S*G must protect its buffer space against a particular local or remote process "running away" and flooding it with data.

To protect its buffer space, M*S*G applies a flow control algorithm separately for each local process. This algorithm depends upon three integers, called N1, N2, and N3, which are essentially limits on the depth of queueing of specifically-addressed messages by M*S*G. Whenever one of these limits is exceeded, M*S*G staunches the flow of messages from the sending process by sending MESS-HOLD or MESS-REJECT replies (to a remote process) or "quenching" the MSG channel (to a local process).

* A local process generating messages faster than they are being accepted by remote MSG's will have an increasing pending-event set. When the number of pending SendSpecific operations exceeds N1, M*S*G will "quench" the process, i.e., reduce its freedom to communicate with M*S*G. Specifically, a quenched process is allowed to issue only receive operations; no other type of MSG request is allowed. Hence, a process may have at most N1+1 pending SendSpecific operations.

  A process remains quenched until at least one of its pending SendSpecific messages has been acknowledged by either MESS-OK or MESS-REJECT, reducing the pending count below N1+1. Note in particular that a quenched process cannot send an alarm or issue a Rescind. The inability to rescind pending SendSpecific operations prevents the

process from freeing itself from the quenched state. The inability to send alarms from a quenched process is actually a violation of a rule for MSG implementation stated in Paragraph 2.8 of [6]. However, the NSW usage of MSG will generally involve simple conversational protocols which will provide a reasonable bound on the maximum number of pending sends that a process will have. Hence, an N1 can be chosen which will not be exceeded by any well-behaved NSW process. Therefore, the inability to rescind messages or send alarms from a quenched process should not be a problem in the NSW.

* When the number of queued SendSpecific messages destined for a particular local process exceeds the integer N2, M*S*G will not accept another message for this process; thus the maximum queue depth for unmatched SendSpecific messages is N2+1 (per process). Initially, M*S*G will defer further messages by returning MESS-HOLD instead of MESS-OK. It will also make a local record of the held message, in the form of a formatted XMIT item. As the process absorbs queued messages, these XMIT items will be sent to the remote process to retrieve the messages.

* If specifically-addressed messages continue to come for a process when more than N3 MESS-HOLD's are outstanding, M*S*G will send MESS-REJECT instead of MESS-HOLD. Thus, the maximum depth of the XMIT queue is N3+1.

The values of N1, N2, and N3 are set as part of the MSG configuration (see the next subsection). However, the choices are governed by the following considerations.

* All memory space for M*S*G buffers and certain dynamic control blocks is taken from a buffer subpool (zone) within the NCP region. In general, this subpool contains (1+P)x2K bytes of memory, if there are P processes materialized.

* Each message queued by M*S*G requires a 16-byte control header plus the inter-MSG protocol item. This means a minimum of 35 bytes for a zero-length message, or 163 bytes for a 1000-bit message (assuming generic codes are used).

* When the N2 limit causes the local M*S*G to send MESS-HOLD's, it uses some buffer space for queuing XMITs for later transmission; each MESS-HOLD requires (at least) 32 bytes from the buffer subpool.

* If the buffer subpool is ever exhausted, M*S*G may behave badly, e.g., terminate and restart, interrupting all conversations in progress. Therefore, it is wise to be conservative about the choices of N1, N2, and N3. Notice that the choices depend upon the maximum probable (or possible) message size. Although M*S*G does not

currently enforce a size limit, it may be desirable to include such a limit as a configuration parameter.

Finally, we note that alarms and generically-addressed messages are not subject to this flow control.

4.4. Local Process Creation

An MSG must be able to create a process automatically in response to a SendGeneric message from a remote process. In the CCN implementation, the created process may be either a batch job under OS/MVT or (more usually) a TSO job executed under control of a virtual terminal created by M*S*G. For each generic class supported at CCN, the configuration tables specify the environment and rules for creating processes in this manner.

When a SendGeneric message arrives addressed to CCN, M*S*G goes through the following steps:

1) If there is a pending ReceiveGeneric from a local process of the desired class, it is matched immediately. Note that the process which issued this receive operation is not required to be in the MSG pool of TSO sessions; it may have been started "manually" from a real terminal, e.g., for debugging. For a match to occur, such a process must have materialized and issued a ReceiveGeneric before the corresponding SendGeneric arrives at CCN.

2) Else, if the ReceiveGeneric specifies qwait= false, a failure reply is sent to the originator.

3) Else, M*S*G attempts to start a process to satisfy the request. The first step is to search the Process Start Table (described below) for the given generic class. The search will be either for a name or a code, whichever is given in the SendGeneric message. If no match is found, a failure reply will be sent to the originating host. The entry, if found, will provide the information needed to start a process either in TSO or in batch.

4) For batch, M*S*G submits a batch job using JCL from a file whose name is determined from the generic name. There is of course an indefinite delay before the batch job begins execution, materializes, and issues a matching ReceiveGeneric.

5) For TSO, a more complex mechanism is required to achieve reasonably fast response. TSO LOGON is subject to significant delays, so M*S*G is designed to maintain a pool of available TSO sessions. The MSG configuration specifies upper and lower bounds for the number of idle sessions within the pool.

M*S*G first attempts to assign an idle session from the pool. If that fails and a configuration parameter specifying the maximum number of sessions is not exceeded, M*S*G issues a new TSO LOGON.

Once a session is available, M*S*G issues a TSO command through the corresponding virtual terminal. The command name is derived from the generic name given.

6) In either case, once the process is started it must materialize and issue a matching ReceiveGeneric. This must happen within a time-out interval (currently 5 minutes), or else the pending SendGeneric will be timed out, and a failure reply will be returned to the remote process.

4.5. Time-Outs in M*S*G

Every message-processing system has the problem of discarding undeliverable messages and recovering their resources when one of the remote agents with which it is communicating goes dead. The only feasible way to handle this in a distributed processing situation is with time-outs. Thus, time-outs are essential, although they are also a great source of difficulty.

There are a number of time-outs defined within M*S*G. Some of these are required in case of remote MSG or host failure; others are concerned with intelligent resource management within M*S*G. The latter class of time-outs attempt to free resources used by idle message paths, implying a delay when such paths are used again. In the present implementation, most of the intervals are built into the code, although it would be better to have them defined by the configuration tables (see next section).

The time-outs currently defined within M*S*G and their present values are as follows:

* Idle ARPANET path to remote MSG -- *Infinity

* Idle (local-process,remote-process) control block (SEQSTRM -- see Section 5.1.5) -- *15 minutes.

* SendGeneric await matching local process materialization -- 5 minutes.

* Remote MSG respond to ICP Request -- 5 minutes.

* Idle User/Server TELNET connections -- 20 minutes.

* Idle direct connection of type 'TCAM' -- 30 minutes.

* Remote MSG respond to CONN-CLOSE -- 5 minutes.

Those times marked with asterisk are determined by the configuration tables and therefore easily changed.

4.6. Configuration

The pragmatics of an MSG implementation include a number of parameters or free variables that must be bound in order to make the local MSG function. This collection of local MSG parameters will be called the "configuration" of the MSG control program.

For example, in order to perform a SendGenericMessage call for a local process that does not specify the target host, M*S*G must consult its configuration table(s) to select a host that supports the given generic process class. The M*S*G configuration must also control the creation (and perhaps deletion) of local processes to answer SendGeneric requests from remote hosts.

There are two primary configuration modules for M*S*G:

* MSGTABC

  that contains the CCN-dependent information required to control the creation and allocation of user processes and for accounting and access control; and

* MSGTABS

  that contains information related to the MSG protocol, including remote addressing and flow-control limits.

For M*S*G implementation efficiency and convenience, tables in both modules include entries that specify the correspondence between MSG full generic names and the one-byte generic codes. These generic-class definitions must be kept in agreement, or the CCN M*S*G will not operate properly.

These modules are coded in Assembler language using macros. The assembled modules containing mixed binary and character text are link-edited into the ICT load module library used for all M*S*G modules. MSGTABS and MSGTABC are marked "not executable" as a safeguard. These modules are transient within the NCP region; that is, M*S*G loads and deletes them as needed -- although MSGTABS is loaded whenever MSGMAIN is not idle.

In addition to the modules just discussed, M*S*G needs OS/360 data sets that contain the TSO procedure lists and Job Control Language (JCL) for BATCH processes. The M*S*G data set names are either defined by their use in OS/360 (e.g., the ICT module library, the PL/MSG public library, and the TSO LOGON procedure library) or derived from the Process Charge Number. These are discussed later.

4.6.1. MSGTABC CONTENTS

The MSGTABC module contains the following information. (The exact format required for this table is given in an appendix.)

4.6.1.1. M*S*G Directory

This is a ten-character string containing the CCN Charge Number and Userid for accounting records written by the NCP for M*S*G.

4.6.1.2. Process Charge Number

This six-character CCN Charge Number is used for all TSO sessions and batch jobs started under M*S*G control. TSO sessions are started with the directory:

<Process Charge Number>.<TSO Userid>

where <TSO Userid> is taken from the Valid Jobname Table.

4.6.1.3. Valid Jobname Table

This table has two functions: (1) validity-checking the jobnames of TSO and batch jobs that attempt to materialize as MSG processes; and (2) defining the "pool" of Userids for starting TSO jobs. The table is logically divided into two sections: the first gives valid jobnames for batch jobs and for TSO jobs that are not started automatically by M*S*G from the pool; the second section specifies the jobnames and implicitly the Userids for the TSO session pool.

Entries in this table are generated by the macro:

JOBNAME <Jobname String>

Here <Jobname String> is a valid jobname, enclosed in quotes. A TSO jobname has the form: 'TSO<TSO Userid>'. The form of a batch jobname is system-dependent; at CCN it is the six-character <Process Charge Number> followed optionally by one or two alphabetic characters. The macro calls specifying TSO pool jobnames must all be at the end of the table.

For the non-TSO-pool jobnames, partial (i.e., prefix) <Jobname Strings>'s may be specified. Whenever a process attempts to materialize, its jobname must match one of the entries in this table; any characters in the actual jobname beyond the Valid Jobname Table entry are irrelevant to the match. Extending this matching rule to its logical extreme, we allow a null jobname entry to match all real jobnames, thus effectively suppressing jobname validity-checking. However, a null jobname must be coded as an asterisk:

```
JOBNAME '*'
```

4.6.1.4. Process Start Table

This table is used by M*S*G to control the automatic creation of processes under TSO and batch. It contains an entry for each generic class for which M*S*G can start a new process. The table is generated by the macro:

```
PROCESS   <Start Control>,
          <Max Process Count>,
          <Generic Code>,
          <Generic Name>,
          <Parm String> (optional)
```

Here the parameters are:

* <Start Control>

  This selects batch or TSO execution of the job. In each case, it gives an additional parameter that controls the job initiation. Specifically, it is a two-character string:

```
<Start Control> ::= B<jnc>
                  | T#
                  | T$
                  | T<blank>
```

  These correspond to the following:

  B<jnc> : A batch job, with jobname of the form:

```
<Process Charge Number><jnc>$
```

  i.e., <jnc> is the 7th jobname character, and the 8th is "wild". The job will be executed with the Charge Number <Process Charge Number>; a Userid, if any, will be taken from the actual JOB card.

T# : A TSO job, with the process started as a Command Procedure. That is, M*S*G will use its virtual TSO terminal to enter the TSO command line:

#<Proc Name> '<Parm String>'

Here <Proc Name> is the first eight characters of the <Generic Name> parameter in this macro.

T$: A TSO job, with the process started as a command processor found in USELIB. M*S*G enters the TSO command:

$<Command Name> '<Parm String>'

Here <Command Name> is the first eight characters of the <Generic Name> parameter in this macro.

T<blank>: A TSO job, with the process started as a command processor found in the STEPLIB defined by the LOGON procedure for this directory. M*S*G issues the TSO command:

<Command Name> '<Parm String>'

* <Max Process Count>

For TSO, this is an integer literal specifying the maximum number of processes (up to 255) that may be materialized in this generic class at the same time. For a batch class, no limit can be enforced by M*S*G, and the character "*" should be coded in this case.

* <Generic Code>

This is an integer in the range 1-127 giving the "generic code", i.e., the internal MSG short form for the generic class name, or '***' if no short form exits.

* <Generic Name>

A character string containing the name of the generic class. The first eight characters of this string are used to select either the JCL member for a batch job or the TSO command/command procedure name.

* <Parm String> (optional)

This is a fixed character string to be enclosed in quotes and included in the command line for TSO; it is not currently supported (i.e., is ignored) for batch jobs.

4.6.1.5. Session Limits

The following four values in the MSGTABC module control the pool of TSO sessions that M*S*G maintains.

* Start-up Session Count

  When it first starts, M*S*G will create this many idle TSO sessions.

* Minimum Idle Session Count

  M*S*G will start a new TSO session whenever the count of idle sessions in the pool drops below this number.

* Maximum Idle Session Count

  M*S*G will destroy excess TSO pool sessions whenever the count of idle sessions exceeds this number.

* Maximum Session Count

  The maximum number of simultaneous TSO pool sessions, either active or idle, that M*S*G will allow.

4.6.2. MSGTABS CONTENTS

There are six information items in the MSGTABS module.

4.6.2.1. MSG Flow Control Parameters

* Quench Limit (N1)

  If the number of pending SendSpecificMessages from a particular local process exceeds the integer N1, M*S*G will quench that local process.

* Hold Limit (N2)

  If the number of messages queued by M*S*G for delivery to a local process exceeds the integer N2, then M*S*G will send a MESS-HOLD reply to any new SendSpecific messages to that local process, and queue an XMIT message for later retrieval of the message.

* Xmit Limit (N3)

  If the number of queued XMIT messages exceeds the integer N3, M*S*G will send MESS-REJECT instead of MESS-HOLD.

* Direct Connection Limit

  This is the maximum number of direct connections that a process can have open simultaneously.

4.6.2.2. Generic Class Table

This table defines the generic classes known to the local MSG. It is generated by the following macro call for each generic class:

```
GENERIC <Generic Code>,<Generic Class>
```

Here, <Generic Class> is a character string giving a generic name (long form), and <Generic Code> is an integer literal giving the corresponding short form.

4.6.2.3. MSG Host Table

This table defines the host address and class of ARPANET hosts that support MSG. For each such host, there is an entry generated by the following macro:

```
HOST    <Net Host Number>,
        <MSG Host Number>,
        <ICP Contact Socket>,
        <Host Type Code>,
        <Authentication Socket>
```

Here, the last argument is the ICP socket to be used to request authentication of the corresponding remote MSG. If <Authentication Socket> is even (e.g., zero), authentication will not be requested.

There is an alternate MSG Host macro, called HOSTN, that takes the same arguments and generates the same entries. It is used to indicate that M*S*G should send an MSG NOP message to this host whenever M*S*G is restarted.

4.6.2.4. Remote Address Table

This table is used by M*S*G to choose a destination host to receive a SendGenericMessage from a local process, based upon the requested generic class. Each entry in the table is the pair:

(<MSG Host Number>, <Generic Code>)

The entries are generated by macro calls of the form:

REMOTE <MSG Host Number>,<Generic Code>

These entries are searched in the order of the macro calls, and the first occurrence of the desired generic code is taken. This means that the CCN M*S*G will always choose the same destination host (except when the process specifies a host).

4.6.2.5. Error Code Table

This table is used to convert MSG protocol reason codes to PL/MSG return codes.

4.6.2.6. Time-out Intervals

Two internal time-out intervals are specified here in units of .Ø1 seconds. Specifying either value as zero effectively sets that interval to infinity, i.e., disables the corresponding time-out.

* SEQSTRM interval

  This is the time to keep an idle process-to-process path queue (SEQSTRM) control block.

* HCT interval

  This is the time interval to keep open an idle host-host connection to a remote M*S*G.

There are no ordering restrictions in any table except that noted under the Remote Address Table.

4.6.3. TSO SESSION CONFIGURATION

To start a TSO pool session, M*S*G issues the following command line:

```
LOGON <userid> ACCT(<Process Charge Number>
                    PROC( NSWMSG[X])
```

The third parameter selects the JCL "LOGON procedure" from one of two members of the system library SYS1.LOGON. The members are called NSWMSG and MSWMSGX, with the latter used by the experimental version of M*S*G. These members contain LOGON procedures by the same names and form part of the M*S*G configuration.

An example of one of these LOGON procedure members is shown in Figure 7 (in Appendix E). Notice that if MSGBUG is to be used, there must be a STEPLIB DD card specifying the library containing the MSGBUG modules. In many cases, the process modules themselves will also have STEPLIB requirements.

The first SYSPROC DD card in Figure 7 defines a user-supplied command procedure library. Members in this library are "canned" sequences of TSO commands that can be invoked by specifying the member name. The most efficient and preferred command procedure invocation uses the 'T#' form of the <Start Control> parameter.

The PARM is set to automatically execute a particular command procedure, called FIRST, immediately after LOGON. This command procedure can invoke the TSO command SETUSE to set the USELIB library. The USELIB library will usually be the same as the STEPLIB library. It can be used to reduce the overhead in command invocation if 'T$' is specified as the <Start Control> parameter.

4.6.4. Batch Session Configuration

To create a batch process, M*S*G must "submit" a data set containing the necessary JCL. It takes the JCL from a member of a library data set named:

<Process Charge Number>.NSW.JCLLIB

The member name used is the first eight characters of the <Generic Name> from the Process Start Table (in MSGTABC). This member must contain the complete job JCL, including the JOB card. The jobname on this JOB card will be ignored, and instead a jobname described earlier will be generated and used. If a STEPLIB is required by the batch job, it should be included in the JCL.

4.7. Test Version of M*S*G

To ease maintenance of the MSG implementation after it is running in production, CCN's MSG has been designed to allow a "test" M*S*G to operate concurrently with the production M*S*G. In general, the test M*S*G will consist of program modules whose names end in "X" (see Section 6). It is not started automatically when the NCP starts, but will start whenever a process explicitly requests the test version. This is accomplished in the following manner:

* From a remote MSG, use the contact socket 27 (decimal) instead of the normal MSG socket 31.

* From a local process, open an Exchange window to the NCP job ('ARPA') with YOURTAG='MSGUSERX' instead of 'MSGUSER'. Normally, this EXOPEN is issued by the process-materialization call of PL/MSG. There is a routine named MSGXPER which the user may call to change PL/MSG's YOURTAG to connect to the experimental version.

5. MSG IMPLEMENTATION DESIGN

5.1. M*S*G Organization

M*S*G is implemented as a set of processes, called "ptasks", which execute within the NCP job. The NCP job executes at the highest dispatching priority in the system, and therefore has maximum priority when calling Supervisor services or seeking resources.

The CCN NCP [10] is a group of programs and service routines executing under control of the Interactive Control Task (ICT). When the NCP job is started, the ICT control program TASKER loads and supervises the NCP programs and service routines. TASKER is a "commutator", that is, it manages a variable set of coroutines, called "pseudo-tasks" or "ptasks", which are considered to be true concurrent processes.

The ICT primitive for forking a new ptask, called "PATTACH", creates an inferior ptask and begins its execution at a specified module entry point. Repeated use of PATTACH by ptasks and sub-ptasks builds a tree of processes in the familiar manner. The "PDETACH" primitive destroys a specified sub-ptask and all of its descendants.

Before M*S*G, the idle NCP contained three ptasks:

* LOGGER -- which "listens" for requests to create Network sessions. These requests may originate either from remote hosts as ICP sequences, or from local user processes as Exchange open requests.

* NCP -- which processes input from the IMP interface.

* IMPIO -- which sends data to the IMP interface.

To handle actual sessions, LOGGER creates sub-ptasks by issuing the PATTACH primitive. LOGGER directly creates "Host Control" ptasks, one for each active host; these in turn create the actual session-handling sub-ptasks.

Two additional permanent ptasks have been added to the idle NCP for M*S*G:

(1) the "main" or Protocol Manager ptask, that manages the MSG protocol, and

(2) the TSO control ptask, that manages a variable pool of TSO sessions in which MSG processes can be started.

Both execute as direct sub-ptasks of LOGGER and are created whenever LOGGER starts.

In addition to the two fixed ptasks, M*S*G creates a variable set of ptasks to handle individual TSO sessions and direct connections, and also transient ptasks which perform specific functions and then terminate themselves ("exit").

The M*S*G code is divided into many modules which are loaded, executed, and deleted from core as needed. The ICT primitive "PXCTL" allows a module to pass control to another module by name, within the same ptask.

5.1.1. MSG Protocol Manager ("Main") Ptask

Whenever the NCP job is started, the LOGGER ptask forks the main ptask as an inferior by issuing PATTACH for the initialization module MSGINIT. This module performs one-time-only M*S*G initialization (e.g., assigning the host incarnation number), uses PATTACH to fork the TSO control ptask, and finally uses PXCTL to transfer control to the main M*S*G module, called MSGMAIN. MSGMAIN, which is always in core, then waits for work. It is awakened, for example, by a message from a remote MSG or by a request from a local process.

MSGMAIN is controlled by two dynamic tables:

* Host Control Table -- contains a set of entries called HCT's, each of which corresponds to an ARPANET connection pair to a remote MSG.

* Process Control Table -- contains a set of entries called PCT's, each of which corresponds to a particular local MSG process.

5.1.2. Incoming MSG Connection Requests

The LOGGER process of the CCN NCP handles incoming requests for connections, either local (i.e., through the Exchange) or from a remote host (i.e., through ICP). Each such request uses a specific Exchange window or "well-known" socket. From the NCP configuration tables, the LOGGER determines the module to be executed to service each such request, and causes PATTACH to be issued to create a sub-ptask executing this module (through the agency of an intermediate Host Control ptask).

In particular, when an incoming request is for M*S*G, the LOGGER starts a transient ptask for either the module MSGUSER (for a local process wanting to materialize) or else the module MSGSRVR (for a remote MSG wishing to

connect to M*S*G). The transient module builds an appropriate HCT or PCT, respectively, for the requested service. If the request is valid (i.e., either the local process is executing in a known job or the remote MSG can be authenticated) and if there are sufficient resources to service another process or host, the new table entry is chained to the MSGMAIN work area. The transient module, having performed its function, then exits and vanishes.

After creating a table entry for a valid incoming request, the transient module may discover that the main ptask does not exist, i.e., it has terminated (presumably abnormally). In this case, the transient module, instead of exiting, transfers control via the ICT PXCTL primitive to an alternate entry point in the M*S*G initialization module; thus, the transient ptask becomes the main ptask.

5.1.3. Outgoing MSG Requests

If a local process sends an inter-MSG protocol item to a host for which M*S*G currently has no HCT (i.e., no open ARPANET connection), the main ptask requests the LOGGER to initiate an "outgoing" ICP request. This is similar to the incoming request except for socket manipulations which the LOGGER performs. MSGMAIN passes to LOGGER the module name MSGUICP for which a transient ptask should be created to service the outgoing request.

This module creates the HCT just as if the request had been incoming, except that: (1) an inter-MSG SYNCH item must be created and chained as the first message to be sent to the remote host, and (2) no authentication operations are necessary.

5.1.4. Authentication Replies

An outgoing request may cause the remote host's M*S*G to authenticate the ICP from the local (CCN's) M*S*G. The authentication will appear to the local M*S*G as an ICP to a specific socket. The LOGGER will issue PATTACH for the authentication module MSGVRFY to handle this request. MSGVRFY will find the MSGMAIN ptask, search the work queues, send the socket information to the remote M*S*G, and finally exit. Note that it is not necessary for MSGMAIN to know that authentication requests exist, because they are serviced entirely by transient ptasks.

5.1.5. MSG Data Movement

The MSGMAIN module, executing on the main (Protocol Manager) ptask, is responsible for the manipulation of all host-to-host MSG data, and is the largest and most complex M*S*G module.

Basically, each state which an MSG protocol item can assume (e.g., in transit to/from a remote host, waiting for reply, or held in response to MESS-HOLD) is represented by a FIFO queue. MSG imposes two constraints on this FIFO structure:

* ALARM items -- must be given priority over all other MSG protocol items. This is accomplished by treating the FIFO queues as LIFO for each ALARM item.

* Special Handling -- requires a special queue at the local process level for each active process-to-process communication path, i.e., each (local-process, remote-process) pair. Each SendSpecificMessage item is first put into such a queue. If any active items exist (e.g., a MESS item for which an acknowledgment has not been returned), there may be Special Handling flags, in either the active or waiting MESS items, that inhibit the flow of some waiting MESS items in this queue. When all conditions have been satisfied, a MESS item is moved to the next queue. Note that some MESS items may be moved past an item which is delayed in the special queue.

5.1.6. Direct Connections

Direct connections are handled by the MSGMAIN ptask and a variable set of inferior "connection control" ptasks. Anchored in each Process Control Table is a chain of direct-connection control blocks, each representing the state of an active connection for this process. One of these control blocks is built when the first CONN-OPEN item is received, and it is destroyed when it is no longer needed (usually after CONN-CLOSE items have been exchanged with the remote MSG).

Each connection control ptask executes the module MSGCONN. It is responsible for allocating the local socket space, opening ARPANET connection(s) to the remote host and an Exchange path to the local process, and transmitting the information from one to the other. When the control ptask is done, it simply exits. Alternatively, the main ptask may issue PDETACH for a control ptask which is no longer needed (e.g., because of time-out waiting for CONN-OPEN from the remote MSG).

5.1.7. Starting a Process

When M*S*G receives a SendGeneric message from a remote process for which a local process must be started, MSGMAIN creates a transient process-start ptask executing the MSGSTRT module. By examining the MSGTABC configuration tables, MSGSTRT determines how to start the required process (e.g., in batch or TSO). If the process

can't be started, an appropriate MESS-REJECT item is returned.

For a batch job, the process-start ptask issues the CCN job submittal SVC. For a TSO session, a TSO command buffer must be created and passed to the TSO control ptask. If no TSO control ptask is found, the process-start module passes control to the TSO control module MSGTSOC, thereby becoming the TSO control ptask.

5.1.8. TSO Session Control

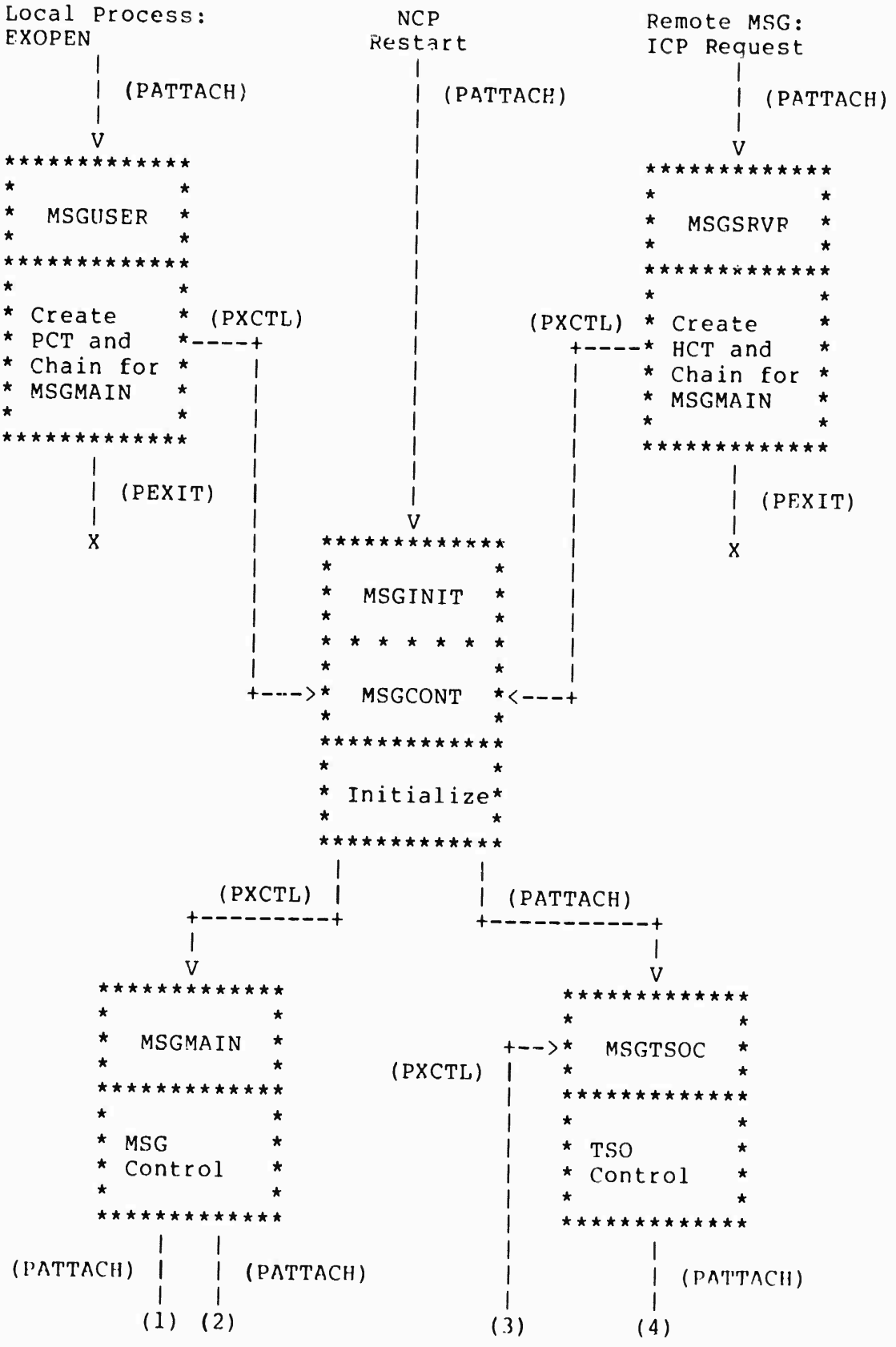
The TSO session pool is controlled by the permanent TSO control ptask, which executes module MSGTSOC, and a variable set of session-control sub-ptasks, one for each TSO session. The session-control ptasks execute the module MSGTSOS.

The TSO control module MSGTSOC is responsible for starting and stopping ptasks for TSO sessions and for assigning work (i.e., starting a process) to an idle session when requested by a transient process-start ptask (discussed earlier). The MSGTABC configuration module includes the minimum and maximum number of idle sessions well as the maximum number of sessions to allow. MSGTSOC creates inferior session-control ptasks, as needed, to start TSO sessions for processes. Conversely, as the processes terminate and leave the sessions idle, the TSO control ptask issues PDETACH when the maximum idle number is exceeded.

Each TSO session-control ptask acts as a TSO pseudo-user, connecting to the TSO terminal I/O-driver job TCAM through Exchange and simulating a virtual terminal. Initially, a session-control ptask must find a free TSO Userid (directory) from the list in the Valid Jobname Table, and use it to log onto TSO. After LOGON is complete, a session-control ptask informs its superior, the TSO control ptask, that it is idle, and then it waits for work (i.e., a request to start a process). Whenever its process terminates, a session-control ptask again informs its superior that it is idle.

When it receives a TSO command buffer from a transient process-start ptask, the TSO control ptask (MSGTSOC) passes the buffer to an idle session-control ptask (MSGTSOS). The latter passes the command as input to TCAM-TSO, which then executes the command, thereby starting the process.

The following diagram illustrates the relationship among the M*S*G modules just described.


Local Process:
EXOPEN
(PATTACH)
MSGUSER
Create PCT and Chain for MSGMAIN
(PXCTL)
(PEXIT)
NCP Restart
(PATTACH)
MSGINIT
MSGCONT
Initialize
Remote MSG:
ICP Request
(PATTACH)
MSGSRVR
Create HCT and Chain for MSGMAIN
(PXCTL)
(PEXIT)
(PXCTL)
(PATTACH)
MSGMAIN
MSG Control
(PATTACH)
(PATTACH)
(1)
(2)
MSGTSOC
TSO Control
(PXCTL)
(PATTACH)
(3)
(4)

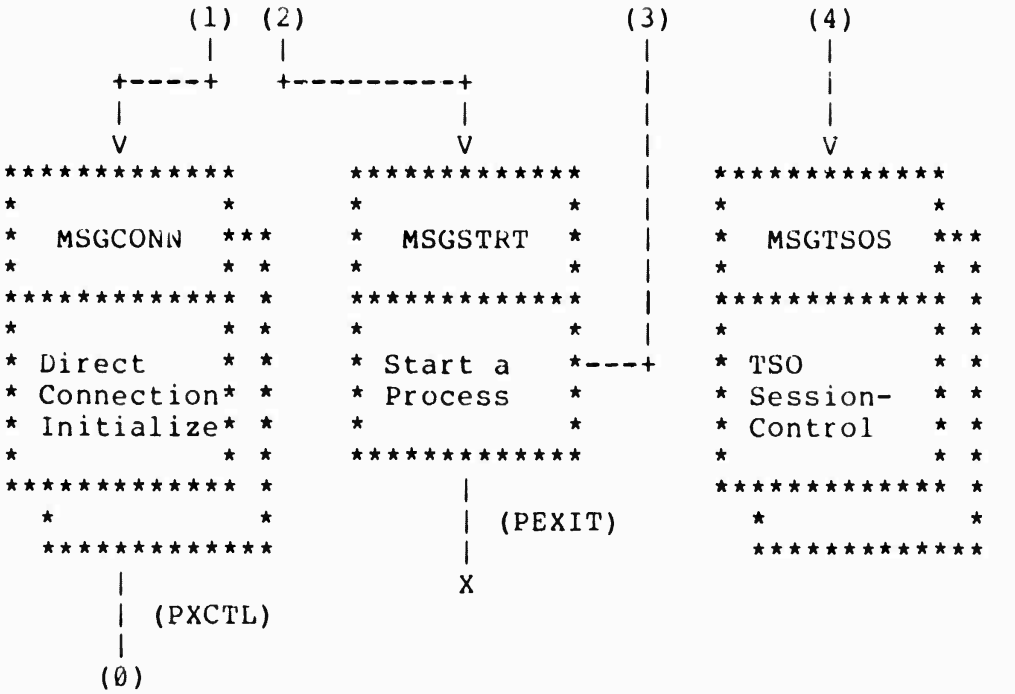


Figure 1. M*S*G Flow Diagram.

5.2. Exchange Protocols

M*S*G uses the Exchange [9] to communicate with its processes and with TCAM-TSO. MSG processes use MSG via the PL/MSG access method and therefore need not concern themselves with the Exchange protocols used over these paths.

The Exchange protocols will be described in this section. Each materialized MSG process uses one primary Exchange window for NSW primitive communication and a secondary window for each direct connection.

5.2.1. Primary MSG Window

The LOGGER always has an Exchange open ("EXOPEN") request pending for a primary Exchange window to a process. To materialize as an NSW progcess, a program issues a matching EXOPEN. When the match occurs and a new primary MSG window is opened, LOGGER creates an MSGUSER ptask, passes the open window to it, and then issues a new pending EXOPEN.

A primary window has five channels (numbered 0 to 4). The EXOPEN request from PL/MSG has the form:

```
EXOPEN  MYTAG='MSGU',
        YOURTAG='MSGUSER[X]', YOURJOB='ARPA'
```

This requests MSG access (i.e., either production or test M*S*G) and forces connection to the NCP job ('ARPA').

After the primary window opens, PL/MSG issues an EXCH request to pass the generic name of the materializing process to the MSGUSER module of M*S*G. MSGUSER responds with the specific process name; in both cases, channel 0 is used. After this identification hand-shake, all further use of the primary window is by MSGMAIN for primitive requests.

Channel 0 is used to notify the process of the completion of any request. The channel is half-duplex, always passing information from M*S*G to the process. The data is always six bytes, four bytes of post code followed by the half-word request id number.

Channel 1 is used to request some action of M*S*G. The channel is half-duplex, always passing information from the process to M*S*G. The data format is variable, depending on the request type; see Figure 3.

Channel 2 is used only for ReceiveGenericMessage. When the process wishes to receive a message generically, it sends M*S*G a time-out interval on channel 2 and waits for the EXCH to complete. When a generic message is available or the interval has expired, M*S*G completes the EXCH by replying with the message information.

Channel 3 is used only for ReceiveSpecificMessage, in the same manner as channel 2. The separate channels are required because an MSG process must be able to receive a generically-addressed or a specifically-addressed message at any time.

Channel 4 is used only for EnableAlarm, in the same manner as channels 2 and 3.

All the data transfers are in MOVE mode on the M*S*G side, and M*S*G uses only ECB signaling on all channels. PL/MSG uses MOVE mode on all channels except 1, where STREAM mode must be used. PL/MSG uses interrupt signaling on all channels except 1, to post the user's ECB and to adjust string lengths when necessary.

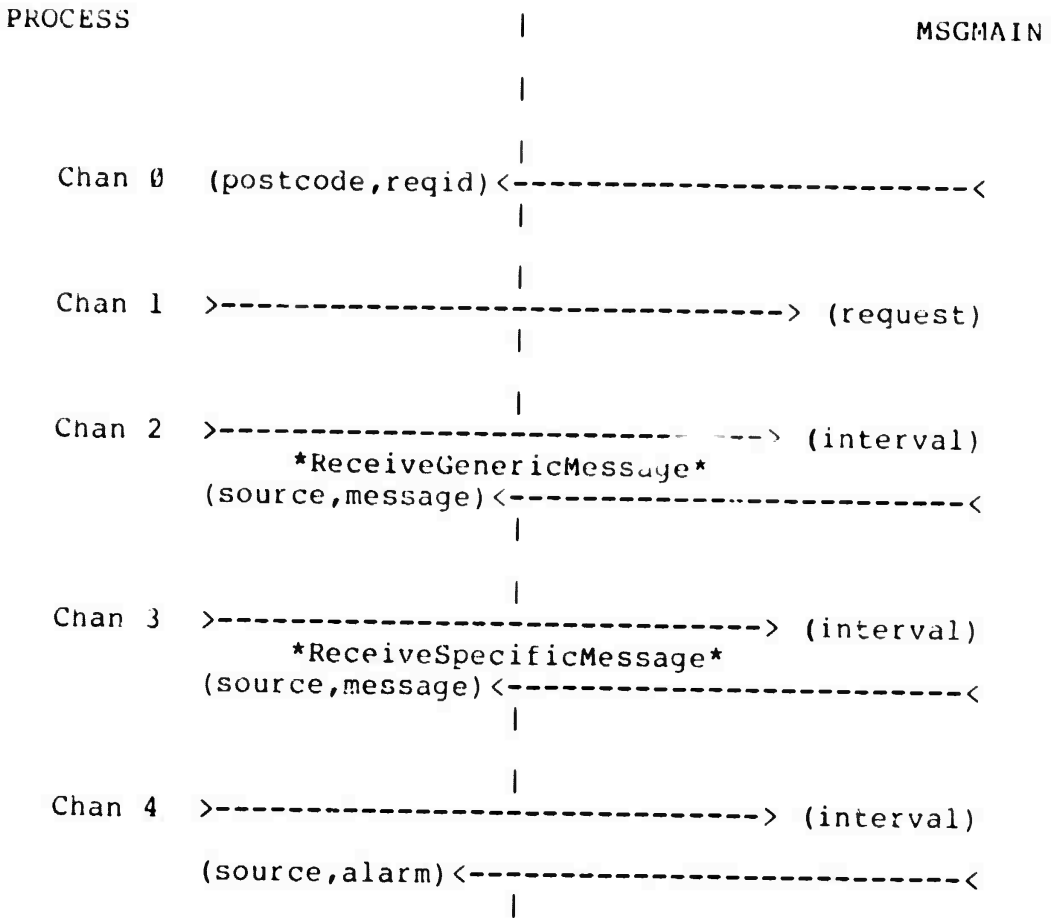


Figure 2. MSG Exchange Protocol

```
Chan 0:   Process  <--  M*S*G
          AL1(x'40',REQCODE),AL2(comp)    ECB Post Code
          AL2(REQID)                      Request Id

Chan 1:   Process  -->  M*S*G
          AL1(REQCODE)                    Request Code
              2=SGM    4=SSM    6=SA     7=CO
              8=TS     9=AA    10=RSNC  11=RSND
          AL3(REQINFO)                    Request Information
          AL4(REQDT)                      Time-out Interval
          AL2(REQID)                      Request Id
          AL2(REQUCID)                    User Connection Id
          AL2(REQHOST)                    Destination Host
          AL2(REQHOSTI)                   Host Incarnation
          AL2(REQINSTN)                   Process Instance
          AL2(REQGNMLN)                   Generic Name Length
          CL127(REQGNAME)                 Generic Name
          CL?(REQMESS)                    Start of Message

     REQINFO depends on REQCODE:
          AA    - AL3(0,0,iaccept)        Alarm Accept Flag
          RSND  - AL3(X'FF',0,RPLCODE)    Receive Rescind Code
                - AL1(0),AL2(REQID)       Request to Rescind
          SGM   - AL1(0,0,waitno)         Will Not Wait
          SSM   - AL1(0,0,handling)       Special Handling
          SA    - AL1(0),AL2(alarm)       Alarm Code
          CO    - AL1(0,type,byte)        Type and Byte Size

Chan 2, 3, and 4:   Process  -->  M*S*G
          AL4(RPLDT)                      Time-out Interval

Chan 2, 3, and 4:   Process  <--  M*S*G
          AL1(x'40',RPLCODE),AL2(comp)    ECB Post Code
              1-RGM    3-RSM    5-EA
          AL2(RPLHOST)                    Source Host
          AL2(RPLHOSTI)                   Host Incarnation
          AL2(RPLINSTN)                   Process Instance
          AL2(RPLGNMLN)                   Generic Name Length
          CL127(RPLGNAME)                 Generic Name

     and one of:
          CL?(RPLMESS)                    Message (RGM)
          XL1'RPLHNDL',CL?(RPLMESS)       Handling Flags, Message
          AL2(RPLALARM)                   Alarm Code (EA)
```

Figure 3. MSG EXCH Data.

5.2.2. Direct Connections

In the EXOPEN request for a direct connection, PL/MSG uses symbolic tags formed from the connection "type" and "connection_id" as the string: "ttttthhhh". Here, "tttt" is one of the mnemonics TCAM, FULL, SEND, RECV, STEL, or UTEL, and the "hhhh" is the hexadecimal representation of the "connection_id" supplied by the user process. The same string is used for both YOURTAG and MYTAG parameters to the EXOPEN.

Once the window is open, a direct connection uses an Exchange protocol that depends upon the connection type. The protocols used for TELNET and for Virtual 2741 ("TCAM") connections are standardized within the CCN system and are documented elsewhere; however, for convenience we repeat these definitions below.

5.2.2.1. Binary Connections

The Binary protocols all use a window with two half-duplex channels. Channel 0 transfers data from the process to M*S*G, and channel 1 transfers in the other direction. A simplex Binary connection is actually a full-duplex connection with one channel unused.

The data transferred across either channel is always defined to be a "bit stream" aligned on a byte boundary. M*S*G will shift and buffer all data to ensure that it only gives multiples of "connection byte-size" to the NCP and integral bytes to the process. It is the responsibility of the process to parse and recombine the bit-streams when connection byte-size is not a multiple of eight.

Channel 0 uses STREAM-to-STREAM mode to send data to the process, and channel 1 uses STREAM-to-MOVE mode to send data to M*S*G. M*S*G uses only ECB signaling on both channels.

PL/MSG requests ConnClose by simply closing the direct connection Exchange window; this also happens if the process abends. In the particular case of simplex RECV connections, M*S*G must issue a dummy EXCH request for channel 0 to determine when the window is closed and hence ConnClose is to be executed. In all other cases, M*S*G will discover the window is closed the next time it tries to send data to the process.

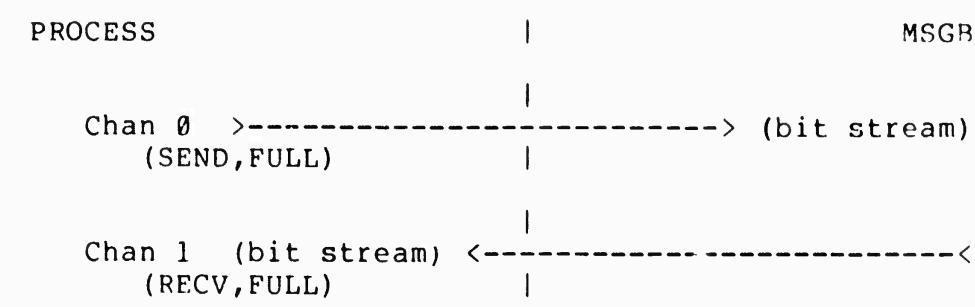


Figure 4. Binary Direct Connection Exchange Protocol

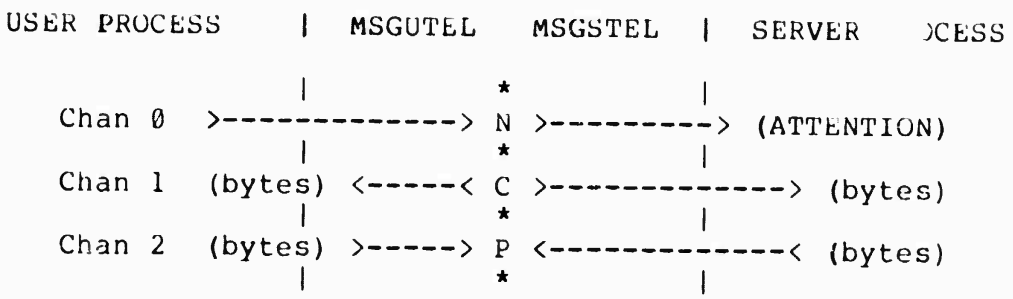


Figure 5. TELNET Exchange Protocols

5.2.2.2. TELNET Connections [15]

The Exchange protocols for User and Server TELNET connections both use three half-duplex channels in the window. Channel Ø is used for signaling an Attention (i.e., an out-of-band signal sent from a User process to a Server process via the TELNET path). Channels 1 and 2 are used for data transfer.

The data transferred on channels 1 and 2 are EBCDIC bytes. M*S*G passes these data bytes between the process and the User/Server TELNET routine in the NCP, without processing the data in any way.

The EXCH requests on channel Ø are half-duplex using LOCATE mode, although the data actually transferred is ignored. A User process must issue an EXCH request to send an Attention, and a Server process should always have a pending EXCH that will complete when an Attention arrives. PL/MSG and M*S*G use interrupt signaling on their respective side of the channel to receive Attentions.

Channel 1 is used to transfer data from M*S*G to the process in STREAM-to-MOVE mode. The data transfer is conversational (i.e., in every EXCH request there is both a message and a reply) with the reply from PL/MSG being a LOCATE-mode synchronization marker. The EXCH requests are interleaved, so that it takes two from one side to satisfy each EXCH request from the other side. Conversational protocol is necessary to handle the "Goahead" signal of the new TELNET protocol -- the TELNET routines must know if a process is ready for more data.

Channel 2 is a simplex STREAM-to-MOVE mode path, that is used to transfer data from the process to M*S*G.

5.2.3. Virtual 2741 Protocol [16]

The Virtual 2741 Exchange protocol, also known as the "TCAM" protocol, is used to create and control a TSO session from M*S*G. The protocol was written to simulate an IBM 2741 terminal and is therefore based on a half-duplex device. This forces the data path to be in either an output or an input state (relative to TCAM) at any given moment.

A communication path is opened to TCAM by issuing an EXOPEN with MYTAG='2741', YOURTAG=*, and YOURJOB='TCAM'. When the EXOPEN completes, the YOURTAG will have been updated to contain the port number in use. Two channels are used in this protocol -- channel 0 is used for data transfer and channel 1 is used to signal Attention (when the data channel is in output state).

By convention, the data channel is in input state when the window first opens (i.e., ready for a LOGON line). To signal that the channel should switch states, the last character transmitted must be X'FF'. To signal an Attention when the channel is in input state, X'FF' (alone) is sent on the data channel. M*S*G takes the responsibility for appending the X'FF' to input lines when necessary.

The data transmitted is always in EBCDIC with any necessary format effectors (e.g., NewLine is needed as a line terminator). The output data from TCAM may have idle characters (i.e., X'FE') which must be removed. Since the idle character is conveniently also a TELNET NOOP command, the NCP TELNET routines remove it.

The channel 0 EXCH request in both input and output states is always STREAM-to-STREAM. The input EXCH request will complete when TCAM has accepted the data. The output EXCH request completes either when TCAM has completely filled the buffer or when TCAM forces a short buffer to complete because the channel must change states.

The channel 1 EXCH request is always in LOCATE mode. It is exactly like the channel 0 request of a Server TELNET process.

6. M*S*G PROGRAM LOGIC

This section describes in detail the program logic of M*S*G, the control program for the NSW interprocess communication mechanism MSG. Familiarity with OS/MVT, the CCN NCP, and the Exchange is assumed.

6.1. M*S*G Environment

M*S*G is executed by a set of processes, called ptasks, within the NCP job and under control of the ICT (Interactive Control Task) commutator.

To implement MSG, the NCP's LOGGER module has been changed to create the main M*S*G ptask, executing the M*S*G initialization module MSGINIT, when the LOGGER is first entered after an NCP restart. If M*S*G later encounters an unrecoverable error, it will terminate. The next MSG request from either the ARPANET or from a local process will cause the main ptask to be recreated and the initialization module to be reentered.

Both a production and a test version of M*S*G can be in operation simultaneously. The test version is not started automatically, but can be started by explicit request from a remote or local process.

The next three sections list the system services that M*S*G uses. Although many services listed use nested calls of other services (e.g., PCORE uses REGMAIN and ATOPEN uses AOPEN), only those services called directly are listed.

6.1.1. OS Services

Very few of the supervisor services (SVC's) of the host operating system OS/MVT are used directly by M*S*G. In the ICT environment [16], many of the SVC's are replaced by ICT service calls (e.g., REGMAIN by PCORE, LOAD by PLOAD), and a ptask is required to use the ICT service to preserve job integrity. Following is the list of the SVC's which M*S*G does use:

* ENQ -- used in the TSO LOGON sequence to test for the OS/MVT ENQ's on the TSO Userid and find a free LOGON directory.

* EXCH -- used to communicate with PL/MSG and TCAM-TSO [9].

* ICT -- used to request an M*S*G error WTO message.

* MEMO -- used in the TSO LOGON sequence [18].

* REGMAIN -- used to get and free M*S*G buffer elements. Note: Use of REGMAIN violates ICT conventions; however, since M*S*G uses a separate core subpool, which it manages, job integrity is preserved. The production M*S*G uses subpool 2, while the test M*S*G uses subpool 3 for this purpose.

* SRS -- the CCN System Routing Service SVC is used to submit jobs to start batch processes. See [19] for SRS details.

* SSVC -- the CCN Service SVC is used to create and delete a CCNUID (user identification block) for the TSO LOGON sequence. See [20] for full details concerning SSVC's.

* TIME -- used to request the time of day for the MSG host incarnation number.

* TYPE 1 -- used in M*S*G initialization to change the ICT environment. This is a CCN SVC that allows a privileged caller to change core not belonging to the caller.

* WTO -- used to display debugging information.

6.1.2. ICT Services [16]

M*S*G uses many of the ICT service routines, called "P-services". Following is the list of P-services used:

* PATTACH -- creates (forks) a sub-ptask.

* PCORE -- obtains and frees core in the NCP region. PCORE is used for Process Control Table (PCT) entries, Host Control Table (HCT) entries, and work areas, but not message buffers.

* PDELETE -- deletes a module from core.

* PDETACH -- terminates a sub-ptask.

* PEXCLOSE -- closes a local Exchange communication path.

* PEXIT -- terminates the issuing ptask. PEXIT will always "clean-up" for the exiting ptask by issuing PDELETE, ACLOSE, etc., when necessary.

* PEXOPEN -- requests a local Exchange communication path, called a "window".

* PLOAD -- loads a module into core (e.g., MSGTABC, which contains the CCN-dependent configuration tables).

* PPOST -- signals to another ptask the completion of an event (e.g., the reception of the second CONN-OPEN causes a PPOST of the MSGCONN ptask).

* PSPIE -- specifies the Program Interrupt Exit address.

* PSTAE -- specifies the Task Abend Exit address.

* PSTATUS -- sets ICT ptask privileges.

* PTASET -- sets the ptask state flags.

* PTATEST -- tests the ptask state flags.

* PWAIT -- relinquishes control to the commutator while waiting for an event to complete.

* PXCTL -- transfers control from one module to another on the same ptask.

6.1.3. NCP Services [21]

M*S*G uses most of the NCP service routines, called "A-services". Following is the list of the A-service routines used by M*S*G:

* AACEBUY -- allocates a socket subspace, called a "port", generally associated with an NCP session. Creates an Account Control Element (ACE) for the session. Used by M*S*G to set up a socket subspace for a direct connection.

* AACESELL -- frees an ACE and its corresponding socket subspace. Used to free the ACE associated with an inter-MSG ARPANET path when an HCT is deleted.

* AALLOC -- sends a host-host allocation for more data.

* ACLOSE -- closes an ARPANET connection.

* AGHCT -- requests the creation of a Host Control ptask for an ARPANET connection.

* AINT -- sends an out-of-band TELNET signal.

* ALSTN -- "listens" for an ARPANET RFC.

* AOPEN -- opens an ARPANET connection.

* ARLSE -- frees space in the circular buffer attached to an open ARPANET receive connection.

* ASEND -- sends an ARPANET message.

* ATGET -- requests a TELNET message or message portion.

* ATOPEN -- establishes a TELNET connection pair.

* ATPUT -- sends a TELNET message.

6.2. M*S*G Modules

All the M*S*G modules (but not the PL/MSG package) are fetched from the same library as the NCP modules, as required by ICT. All M*S*G module names begin with the three characters "MSG" and are seven or eight characters long (because of OS/MVT restrictions). The fourth through seventh characters in each module name identify the particular module. The eighth character, when present, is always "X" and indicates that the module belongs to the "test" M*S*G.

The following list contains all M*S*G modules in alphabetical order. The approximate sizes, in K (1024)-byte units, are also shown in parentheses.

* MSGBINP

  The interface module for direct connections of type Binary. It is used for both full and half duplex. Both production and test systems use the same module name. (0.9K)

* MSGCONN[X]

  The initialization module for direct connections. It performs initialization by connection type and then issues PXCTL to transfer to the appropriate interface module. (2.1K)

* MSGCONT[X]

  An alternate entry to MSGINIT, used by MSGSRVR when there is no active main M*S*G ptask.

* MSGINIT

  The M*S*G initialization module. This primary entry is used only by the LOGGER, through PATTACH. (0.7K)

* MSGMAIN[X]

  The main M*S*G module, the Protocol Manager. MSGMAIN is entered from MSGINIT or MSGCONT[X], and is executed as a sub-ptask of LOGGER. (11.8K)

* MSGSRVR[X]

  The incoming logger module that handles ICP requests from remote MSG's. MSGSRVR opens the connections and creates a HCT. It may also request authentication of the remote M*S*G. (1.4K)

* MSGSTEL[X]

  The interface module for direct connections of type Server TELNET. (0.4K)

* MSGSTRT[X]

  The process-start module executed as a sub-ptask of MSGMAIN to service a SendGenericMessage which requires a process to be started. (1.0K)

* MSGTABC[X]

  Contains the CCN-dependent M*S*G configuration data and tables. It is loaded and deleted as necessary.

* MSGTABS[X]

  Contains the MSG-dependent configuration information. It is loaded and deleted as necessary.

* MSGTCAM[X]

  The interface module for Server TELNET direct connections of special type "TCAM". This connection type is supported at CCN to facilitate "encapsulation". (0.6K)

* MSGTSOC[X]

  The TSO control module. This module is entered by a PATTACH issued from the main ptask by MSGINIT or MSGCONT[X], but it moves itself up the ptask tree to be a direct sub-ptask of LOGGER. (1.0K)

* MSGTSOS[X]

  The TSO session module, executed by a sub-ptask of the MSGTSOC[X] ptask. (1.2K)

* MSGUICP[X]

  The outgoing logger module that handles requests to ICP to a remote MSG and creates an HCT for it. (0.8K)

* MSGUSER[X]

  The module that handles materialization requests from local user processes. It is started by LOGGER as a result of a primary Exchange window opening, and it creates a corresponding Process Control Table entry (PCT). (1.4K)

* MSGUTEL[X]

  The interface module for direct connections of type User TELNET. Note that this is the only M*S*G system module that is not exclusive to M*S*G; it is also used by CCN's User TELNET interface UTELNET. (0.4K)

* MSGVRFY[X]

  The incoming logger module that is given control when a remote M*S*G wishes to authenticate CCN's MSG. (0.5K)

* IGC65ICT, IGC66ICT, IGC67ICT

  Three loads of the ICT SVC, used only by the M*S*G system. They are fetched from the SVC library and invoked whenever an M*S*G error WTO (Write To Operator) message is necessary.

6.3. Control Blocks and Work Areas

M*S*G builds and uses many control blocks in core that is obtained either by PCORE or by REGMAIN in the M*S*G subpool. In addition, M*S*G uses some ICT and NCP control blocks that are built either explicitly or implicitly by an M*S*G request.

6.3.1. ICT Control Blocks

* COMWORK

  is the main ICT commutator work area and contains the anchors for all ICT chains. When a ptask is dispatched, COMWORK also contains the time-of-day which M*S*G needs for time-out calculations.

* PTA

is a Pseudo-Task Area, the combined ICT control area and private work area for a particular ptask. A PTA is created by PATTACH and destroyed by PEXIT. In addition to all information needed by the ICT commutator to run the ptask (e.g., status flags, waiting event set, and saved registers), the PTA has fields reserved for the application. See Figure 6 for MSGMAIN's use of its PTA.

* PCLIST

is the ICT dispatching list, a chain of entries anchored in COMWORK and pointing to the PTA's. Transient M*S*G routines use this chain to find the MSGMAIN and MSGTSOC ptasks. PCLIST entries are created and destroyed implicitly by PATTACH and PEXIT.

* QEL

represents an area of core obtained by calling PCORE The chain of QEL's is anchored in COMWORK.

* XEL

represents an Exchange window opened by PEXOPEN. The chain of XEL's is anchored in COMWORK.

M*S*G uses the QEL and XEL chains only when MSGCONN must "steal" the core and Exchange window from an MSGTSOS-started process in order to open a direct connection of type TCAM.

6.3.2. NCP Control Blocks

* ACE

An Account Control Element corresponds to a single NCP session and therefore a socket subspace ("port"). The ACE contains CCN charging and host-to-host control information. One exists for each active inter-MSG connection and each direct connection. M*S*G sets the appropriate charging fields and uses the status flags to determine whether debugging WTO's should be generated for an HCT and whether the operator has requested that the path be aborted.

* CCB

The Connection Control Block is chained from the ACE and represents an open ARPANET connection. It contains the data needed by the NCP to service the connection. M*S*G uses the CCB to check the status

of the connection and retrieve data as it becomes available.

* TCCB

  The TELNET Connection Control Block is a CCB with the additional fields needed by the NCP TELNET routines.

* WRE

  The Write Request Element is the parameter passed to ASEND to send a message on an [illegible] connection.

6.3.3. M*S*G Control Blocks

All M*S*G control blocks use the first word as a link to the next entry in the chain, with the value zero indicating the end of the chain. In addition, control blocks that were obtained by REGMAIN use the second word for the length of the control block.

HCT

A Host Control Table entry, or HCT, represents an open communication path to a remote MSG. An HCT contains the following fields:

* The address of the next HCT.

* The address of the ACE.

* The ARPANET-form remote host number.

* The MSG-form remote host number.

* The remote M*S*G's incarnation number.

* Status flags pertaining to the communication path.

* HCTSENDQ: the anchor for the chain of buffers waiting to be sent to the remote MSG.

* HCTSENTQ: the anchor for the chain of buffers awaiting acknowledgment by the remote MSG.

* HCTHOLDQ: the anchor for the chain of buffers "held" and awaiting an XMIT command.

* The address of the current read buffer.

* The addresses of the send and receive CCB's.

* The timer entry used to determine if the inter-MSG connection has been idle too long; if so, an MSG CLOSE can be sent to the remote M*S*G.

* Two WRE's, leader areas, and addresses of data currently being sent.

* The receive-socket circular buffer.

PCT

A Process Control Table entry, or PCT, controls a local MSG process. It contains the following fields:

* The address of the next PCT.

* The local process instance value.

* The termination signal request identification number.

* PCTINQ: the anchor for the chain of items waiting to be sent.

* The anchor for the chain of SEQSTRM control blocks.

* PCTOUTQ: the anchor for the chain of acknowledgments awaiting delivery to PL/MSG.

* The anchor for the chain of CCONTRL control blocks.

* Status flags pertaining to the Exchange path to the process.

* The process's generic code, if any.

* The length of the process's generic name.

* The generic name, in uppercase EBCDIC and padded with blanks.

* The ALARM-accept state.

* The five Exchange control blocks (EXB's) and extent lists required by the PL/MSG interface.

* The time-out timers for the three possible receive operations.

* Anchors for three queues of inter-MSG protocol items from remote processes that are queued awaiting a corresponding receive operation from this process. The queue anchors are named by the type of matching receive operation:

PCTRSMQ -- ReceiveSpecificMessage Queue
PCTRGMQ -- ReceiveGenericMessage Queue
PCTEAQ -- EnableAlarm Queue

Note: When the length of PCTRSMQ exceeds N2, M*S*G will stop accepting messages for this process (by sending MESS-HOLD instead of MESS-OK). It will also start queuing corresponding XMIT items on the PCTXMITQ. Hence, the length of PCTRSMQ cannot exceed N2+1.

* The address of a 2K segment of the M*S*G subpool.

* The count of MESS items delayed (not sent to a remote MSG) in all SEQSTRM blocks for this process. The delay is caused by the sequencing or stream-marking rules.

* The count of SendSpecific messages sent but not acknowledged by a remote MSG.

Note: When the sum of the last two counts exceeds N1, M*S*G will quench the process.

* PCTXMITQ: the anchor for the chain of XMIT items waiting to be sent when the process's PCTRSMQ length drops below N2+1.

Note: When the length of this queue exceeds N3, further specifically-addressed messages to this process will be rejected.

SEQSTRM

Each of the SEQSTRM ("SEQuenced/STream-Marked") control blocks, chained from the PCT, has information needed for "special handling" of MESS items sent by the corresponding local process to a particular remote MSG. It contains the following fields:

* The address of the next SEQSTRM entry.

* The length of the entry.

* The timer value for deletion of an idle entry.

* The anchor for the chain of MESS items delayed by special handling.

* The count of inter-MSG items sent but unacknowledged.

* Status flags pertaining to special handling.

* The destination process's generic code, if any.

* The full specific process name of the destination process.

CCONTRL

Each of these control blocks, chained from a PCT, has information needed to control a direct connection. A CCONTROL block contains the following fields:

* The address of the next CCONTRL entry.

* The length of the entry.

* The timer value for aborting the connection.

* The address of the CONN-OPEN item, when waiting for the local socket to be assigned.

* The PTA address of the connection control ptask.

* The connection control ptask's termination internal ECB.

* The local and remote socket numbers.

* The source and destination message id's.

* The user-connection id.

* The local and remote "reason" codes.

* The local connection type and byte size.

* Status flags pertaining to the connection.

* The remote process's generic code, if any.

* The full specific process name of the remote process.

REPLY

"REPLY" is used in M*S*G as a general term for a buffer area. A REPLY (buffer) is used to hold any transient information, e.g., an inter-MSG item waiting to be sent or received. The first two words are always:

* The address of the next entry in the chain.

* The length of the entry.

The next two words are usually:

* The timer value for the entry.

* The MSG-form host numbers of the receiving and sending M*S*G's.

The remainder of the entry is an inter-MSG or a PL/MSG protocol item.

6.3.4. M*S*G Work Areas

MSGMAIN has three work areas. Two are used to keep event lists. There is one list of all Exchange ECB's, five per PCT, one of which is posted whenever a PL/MSG operation completes. The second list has all internal ECB's, one per connection, one of which is posted whenever a direct connection control ptask terminates. The third work area is used to receive EXCH requests from .

MSGCONN builds work areas with EXCH cont. blocks and WRE's as necessary to service direct connections.

MSGSTRT calls PCORE to obtain a work area in which it builds the TSO command line for MSGTSOC or constructs the SRS parameter lists.

```
PTA       DSECT                    Pseudo-Task Area
          ORG     PTAFLPRS         origin to floating point regs.
MSGMOD    DS      CL8              module for PATTACH/PLOAD
MSGCMWRK  DS      A                address of COMWORK
MSGTIMED  DS      F                time of day at cycle start
MSGNAMET  DS      A                address of MSGTABS
MSGECBLA  DS      A                address of ECB lists addresses
MSGINVOC  DS      H                process instance allocation
MSGPTAFL  DS      0X               flags
MSGBECBL  EQU     X'80'            rebuild ECB list
MSGBICBL  EQU     X'40'            rebuild internal ECB list
MSGCONNM  DS      H                number of open connections
MSGPCTNM  DS      X                number of PCT's
MSGWRKNM  DS      X                pages in work area
MSGECBNM  DS      X                pages in ECB list
MSGICBNM  DS      X                pages in internal ECB list
          ORG     PTAPARM          origin to parameter address
MSGMYHST  DS      H                my MSG form host number
MSGINCAR  DS      H                my host incarnation number
          ORG     PTAUSER          origin to user area
MSGHCTA   DS      F                anchor for HCT chain
MSGCCNQ   DS      F                anchor for local-delivery chain
MSGPCTA   DS      F                anchor for PCT chain
MSGHCTQ   DS      F                anchor for awaiting-HCT chain
MSGPCTQ   DS      F                anchor for awaiting-PCT chain
MSGSHPDT  DS      F                anchor for start-timer chain
```

Figure 6. MSGMAIN Usage of the PTA.

6.4. Detailed Module Operation

Each of the following subsections gives a detailed description of the internals of a particular M*S*G module; the modules are listed in alphabetical order.

In general, when a service routine returns an abnormal condition code, the M*S*G module requesting the service will issue an ICT SVC call with a request for a particular M*S*G error WTO. A minor error will only cause the current procedure to abort. A major error may cause all of M*S*G to abort. See Appendix B for a detailed list of possible error messages and suggested remedies.

6.4.1. MSGBINP

MSGBINP, the interface module for Binary direct connections, is given control by PXCTL from MSGCONN. Before issuing the PXCTL, MSGCONN (1) opens the ARPANET socket(s) and Exchange window between the processes, (2) initializes an input work area (i.e., EXB, extent list, and circular buffer) and/or an output work area (i.e., EXB, extent list, WRE and leader, and data buffer), and (3) creates the EXCH ECB wait list.

When MSGBINP finishes processing, it simply issues PEXIT. This notifies MSGMAIN that the process has issued, or MSGBINP has forced, ConnClose. MSGBINP indicates normal ConnClose by changing the return code in PTACMP which MSGCONN set to indicate transmission error.

The first thing MSGBINP does is to wait for something to happen by issuing (the only) PWAIT. MSGBINP waits for any of: socket close, input available or output complete (to/from the NCP), any EXCH complete, or a twenty minute time-out elapsed without activity. If the time-out elapses, MSGBINP forces ConnClose (with transmission error flagged).

If the process is sending, MSGBINP processes the send stream; otherwise, it skips ahead to receive-stream processing. Send stream processing begins by checking the send CCB (socket) to determine whether it has closed. If the socket closed with data waiting to be sent, MSGBINP forces ConnClose with transmission error flagged. If the connection closed with no data waiting, MSGBINP forces normal ConnClose if the connection type was SEND; however, a connection of type FULL is effectively treated as a RECV connection, and MSGBINP starts closing its receive socket.

If the send socket is not closed, the WRE is checked to determine whether the previous ASEND is completed; if not, processing of the send stream is finisned. If the ASEND is completed, MSGBINP must move any unsent bits (if connection byte-size is not eight) to the beginning of the buffer and issue an EXCH request to move data into the remainder of the buffer (if no EXCH is pending). Otherwise, the ECB in the EXB is checked to see whether the EXCH has completed. If the EXCH request is not completed, send stream processing is finished.

If the EXCH has completed abnormally, MSGBINP forces ConnClose with transmission error flagged. If the window was closed by the process to indicate normal ConnClose, MSGBINP exits with normal ConnClose; or for a connection type of FULL, it effectively becomes a SEND connection to transmit any remaining data.

If the EXCH completed normally, more data may be available for sending. Any new data must first be "tacked on" to the end of the previous remainder. If the resulting bit string is at least one ARPANET byte, ASEND is called to send it; otherwise, an EXCH request for more data is issued.

This complete the processing of the send stream. If the process is receiving, the receive stream is now processed. Otherwise, control returns to the PWAIT again, to await more work. First, however, MSGBINP checks the ACE for operator STOP or CANCEL, either of which force immediate ConnClose with transmission error flagged.

Receive stream processing begins with testing the "dummy" LOCATE-mode EXCH request for completion, if the type is RECV. If this EXCH completed, ConnClose if forced; if there is more data for the process a transmission error is flagged.

If there is a pending EXCH, it is checked for completion. If it is not completed, receive stream processing is finished. If the EXCH completed abnormally, Connclose is forced; if data was available, a transmission error is flagged. However, for type FULL the close is delayed until any remaining send data is sent.

When the EXCH completes normally, the process has received the buffered data, and ARLSE is used to free the space in the circular buffer. Now there is no EXCH pending.

When no EXCH is pending, the receive CCB is checked to determine whether the socket is closed. When the socket is closed and after any last data in the circular buffer has been given to the process, the connection is closed in the same manner described above for an abnormal EXCH return.

If the socket has not closed, MSGBINP checks the amount of data now available in the circular buffer. When there is at least one byte of data, the EXCH extent list is updated to describe the data in the circular buffer and the EXCH request is issued. The ACE is then checked, and MSGBINP issues the PWAIT call.

6.4.2. MSGCONN

The MSGCONN module is used to initialize a direct connection. It is executed by a ptask created by MSGMAIN to handle a single direct connection. MSGCONN first obtains an ACE and a local socket (subspace) and then completes initialization in a manner appropriate to the connection type. It finally transfers control (via PXCTL) to a data-transfer interface module appropriate to the connection type.

When MSGCONN is given control, it uses the AACEBUY routine to allocate a new socket subspace ("port") and build an ACE for the connection. If the ACE is successfully built, the MSGTABC tables are loaded and the "M*S*G Directory" parameters are copied into the ACE. The ACE protocol field is initialized to 'MSGCONN' to indicate that the path is not yet open, and the local socket is copied from the ACE to the CCONTRL block. MSGCONN then issues PPOST USER for the main ptask to indicate that the local sockets have been obtained, deletes MSGTABC, and issues PWAIT USER. When MSGMAIN has received matching CONN-OPEN's from both processes, it issues PPOST USER for MSGCONN, allowing MSGCONN to continue.

When MSGCONN continues, it sets the SYSTEM field of the ACE to the Exchange window tags and changes the PTAPGM field from 'MSGCONN[X]' to 'MSGtttt[X]' where the 'tttt' is the (local) connection type. The rest of the initialization depends on the connection type.

For a Binary connection, the Exchange window is opened. After the matching EXOPEN is issued, MSGCONN buys the input and/or output work areas, initializing the EXB's and the WRE. Next, the ARPANET connection(s) are opened. Once the RFC(s) have been sent but before the connection(s) are known to be open, the ConnClose exit code is changed from 'Resources not available' to 'Transmission error'. After the receive socket (if any)

opens, AALLOC is called to send a host-host allocation. Lastly, the EXCH ECB wait list is built and, if necessary, the dummy LOCATE-mode EXCH request is issued. MSGCONN then calls PXCTL to transfer control to the binary interface module MSGBINP, after setting the PTA wait flags so that the first PWAIT issued by MSGBINP will complete immediately.

For a TELNET connection, the Exchange window is opened. After the EXOPEN is completed, MSGCONN issues a TELNET open request to open the ARPANET sockets and create a TELNET work area. MSGCONN then initializes the EXB's in the work area, issuing the first EXCH on the send-data channel. MSGCONN then builds the EXCH ECB wait list, changes the ConnClose exit code to 'normal', and transfers control to either MSGUTEL[X] or MSGSTEL[X] to handle the direct connection.

For a TCAM connection, MSGCONN scans the PCLIST entries to find the session-controlling ptask whose TSO session issued the ConnOpen. When the PTA for the particular MSGTSOS session is found, the environment for the session is changed so that the next time the PTA is dispatched, it will issue a PEXIT. (Note: MSGCONN cannot PDETACH the MSGTSOS PTA because it is not a direct descendant.) while still on the same commutator cycle (i.e., indivisibly with termination of the MSGTSOS ptask), MSGCONN scans the QEL's and XEL's to find both the work area and the TCAM Exchange window which belong to the MSGTSOS PTA, and changes the "owner" fields in the QEL and XEL to point to itself.

The TCAM window data channel is in output state at this time.

Next, MSGCONN issues the TELNET open primitive to open a TELNET connection pair to the remote process. When the connections are open, MSGCONN opens the Exchange window to the local process. No data is ever transferred by the local process over this Exchange path; it is needed only for synchronization, so that the local process will not issue a TGET (changing the data channel state) before MSGCONN has completed stealing the window. When the process matches the Exchange open, MSGCONN transfers to the interface module MSGTCAM[X].

6.4.3. MSGINIT/MSGCONT

MSGINIT is executed by the main M*S*G ptask, which is created by LOGGER when the NCP is restarted. When it is given control, MSGINIT changes the PTAPGM field to 'MSGMAIN' and sets the host incarnation number and MSG-form host number in fields MSGINCAR and MSGMYHST. It then skips around its alternate entry, MSGCONT[X], which is used by MSGUSER[X] and MSGSRVR[X].

MSGINIT next sets the PSPIE and PSTAE options. It obtains eight bytes in the M*S*G subpool and then frees the entire subpool. This is done in case a previous incarnation abended, leaving garbage in the region.

Next, PCORE is called to get four pages of core to build a one-page, empty, PL/MSG EXCH ECB wait list, a one-page, empty, direct connection interface PTA termination ECB wait list, and a two-page initial work area.

Next, PLOAD is issued for MSGTABS, the configuration tables which are needed while M*S*G is active, and PATTACH is issued for the TSO session control module MSGTSOC. The address of MSGTABS is saved in the field MSGNAMET. MSGINIT then gives up the commutator, for the first time, for 15 seconds to allow LOGGER to complete its initialization functions.

When control is returned, the address of COMWORK is found and saved in MSGCMWRK. Next, MSGINIT buys and chains a PCT (marked closed) to MSGPCTA, incrementing the count in MSGPCTNM. MSGINIT chains on the PCTINQ a NOP item for every remote host in the MSGTABS Host Table whose entry was generated by a HOSTN macro.

All initialization functions having been completed, MSGINIT calls PXCTL to transfer control to MSGMAIN[X] and sets the PTA USER flag so that the first PWAIT in MSGMAIN will complete immediately.

6.4.4. MSGMAIN

MSGMAIN is the module that actually implements the MSG host-host protocol. Each functional operation which MSGMAIN must perform (e.g., accepting a request from a local process or sending an NCP transmission when the path is free) is done in a loop for all existing control blocks representing the function, checking each control block in turn to see whether the function is applicable in the current pass. There are thirteen such closed loops of functional operations. Between each function, MSGMAIN issues a PWAIT RETURN to relinquish the commutator for one cycle.

Each of these functional operations will be described in order in the subsections which follow. When all functional operations are complete and before MSGMAIN branches back to the main PWAIT, it checks to see if M*S*G is idle (i.e., no PCT's, HCT's, or entries in MSGPCTQ or MSGHCTQ). If M*S*G is idle, MSGMAIN deletes MSGTABS and then issues PWAIT USER (only). When MSGUSER or MSGSRVR are given control because a process materializes or a remote host wishes to communicate, a PPOST USER is performed to awaken MSGMAIN. On awakening,

MSGMAIN reloads MSGTABS and branches to the main PWAIT, after setting the USER flag to cause the main PWAIT to complete immediately. The functional operation cycles that service the requests thus begin again.

The first thing MSGMAIN does is to set its three base registers. After these have been set, the main logic loop is entered.

6.4.4.1. Main PWAIT

The first operation in the main logic loop is to rebuild the ECB and/or internal ECB wait lists as necessary. Two flags in MSGPTAFL, MSGBECBL and MSGBICBL, control the rebuilding of the lists. When rebuilding the ECB list, MSGMAIN must include the addresses of the five EXCH ECB's for all PCT's. When rebuilding the internal ECB list, MSGMAIN must include the address of the internal ECB in each CCONTRL for all PCT's. However, the internal ECB list includes only those ECB's which are posted complete or marked "internal" (by a X'20' in the high-order byte), meaning that the connection control ptask is active. A common subroutine is used to free the current list and to obtain a larger one. This subroutine calls PWAIT CORE, which is the only place other than the main PWAIT where control is returned to the commutator to wait for an event. After each list has been rebuilt, the appropriate MSGPTAFL flag is turned off.

Next, MSGMAIN issues its main PWAIT. The field PTATIME, which is used by ICT to contain the time-out time requested in the PWAIT, is used by MSGMAIN to contain the minimum time-out interval. This field is altered by PWAIT only if a time option is specified. MSGMAIN uses two subroutines for all time computations; one subroutine changes an interval to a time-of-day and the other checks for time-out of the interval (i.e., time-of-day greater than value in timer word being checked), and updates PTATIME if a smaller positive interval is checked. Immediately before the PWAIT, PTATIME is changed from interval to time-of-day, and when control is returned, PTATIME is set to the M*S*G maximum wait interval of 5 minutes.

The main PWAIT suspends the MSGMAIN p-task until any of the events:

* INPUT -- an ARPANET message from a remote MSG

* OUTPUT -- an ARPANET message sent to a remote MSG

* TIME -- the time-of-day for the next time-out event

* CLOSE -- any inter-MSG connection closing

* ECB -- any EXCH ECB in the list or any internal ECB (indicating that an interface ptask has terminated, i.e., ConnClose has been issued or forced for a connection).

When any of these events occurs, MSGMAIN is marked dispatchable and allowed to run by the commutator. The first thing MSGMAIN does is to save the current time-of-day found in COMWORK, to be used in all time computations this cycle (i.e., this complete pass through the MSGMAIN functional operations).

6.4.4.2. Action Request

The first functional operation is to scan all PCT's for operation requests from local processes. A PCT is skipped if it is marked as quenched (by flag PCTFQNCD in PCTFLAGS+1). The PCT is also skipped if the EXCH ECB for channel 1 is not posted complete. If the EXCH completed abnormally, PCTFCLSE is turned on in PCTFLAGS to close the PCT, the ECB is cleared, and the PCT is skipped.

If the channel 1 (request) EXCH is completed, the flag PCTFQNCH is tested to see if the PCT should be quenched. If so, PCTFQNCD is turned on, the ECB is cleared, and the PCT is skipped. Otherwise, the EXCH, which specified MOVE mode with zero length, has supplied the total data length. If the current work area is not large enough for this, the common subroutine is called to free the old work area and obtain a larger one. An EXB and extent list is formed in the beginning of the work area, and another MOVE-mode EXCH is issued to retrieve the data, matching PL/MSG's still-pending STREAM-mode EXCH. This EXCH should complete immediately because the PL/MSG side was waiting, so the "PWAIT ECB" that follows is needed only for TSO processes that EXCH must cause to be swappped into core. If this data transfer EXCH completes abnormally, PCTFCLSE is turned on and the PCT skipped.

Now, the work area contains data as described by Figure 3 for channel 1. Before examining the data, however, the program again issues the zero-length EXCH request for the length of the next operation, using the EXB in the PCT. Then, the EBCDIC generic name in REQGNAME, if any, is forced to uppercase. The REQCODE is then used as a branch index to the appropriate service routine.

A bad REQCODE immediately generates a request complete-with-error code. A common routine is called to obtain a REPLY buffer, chain it to PCTOUTQ, and move in the completion code and REQID (i.e., the channel Ø EXCH data). The next PCT can now be processed for an action request.

Depending on the REQCODE, one of the following will happen:

* TS -- TerminationSignal

  The REQID is saved in the PCT. If M*S*G is ever changed to reply to STOPME, this REQID must be used. If more than one TS is issued by the process, only the last REQID is saved.

* AA -- AcceptAlarms

  The IACCEPT flag is saved if it is valid, and a completion code (See Appendix D) of zero is returned. If IACCEPT is invalid, the code indicating BADBITS is returned. The common subroutine is called to obtain and chain a channel Ø REPLY to PCTOUTQ

* RSNC -- Resynch

  The SEQSTRM chain anchored at PCTSSQ is searched for a matching destination process and, if found, the "out-of-sequence" bits are turned off. After the chain is searched, the common subroutine is called to build a channel Ø REPLY with zero completion code.

* RSND -- Rescind

  If a "receive" operation is being rescinded, a REPLY buffer for a RESCINDED completion code is obtained and chained to either PCTRGMQ, PCTRSMQ, or PCTEAQ if a request is outstanding and there is nothing in the queue. If the RSND is not for a "receive" operation, all work queues are searched, and if the request is found, it is deleted, and a reply indicating RESCINDED is chained to PCTOUTQ. After the RSND operations have been performed, a REPLY for the RSND itself is generated.

* SGM -- SendGenericMessage

  If the request is invalid, a REPLY is immediately chained to PCTOUTQ to complete the pending event. If the SGM request is valid, the data is copied into a REPLY and chained to the PCTINQ. The next PCT is then processed.

* For SSM, if the request is invalid, a REPLY is immediately chained to PCTOUTQ to complete the pending event. If the SSM request is valid, the data is copied into a REPLY buffer which is chained to the SEQSTRM entry (which is built if none existed). The copying and chaining of the message for SGM and SSM is done by a common subroutine which builds an MSG MESS item in the REPLY buffer.

* SA -- SendAlarm

  A SA is processed the same as a SGM, except that the alarm is chained to the head of the queue, and an inter-MSG ALARM item is built in the REPLY buffer.

* CO -- ConnOpen

  An invalid CO is immediately rejected with a REPLY on PCTOUTQ. The data in a valid CO is copied into a CONN-OPEN MSG item in a REPLY buffer. The CCONTRL blocks are then searched for a matching entry. If the entry is found and if it was created by a previous CO, this CO is invalid and is rejected. If the entry is not from a previous CO, it must have been built by a remote CONN-OPEN. Since both CONN-OPENS are now available, the two requests are checked for parameter mismatches and both marked for CONN-CLOSE or CONN-REJ if necessary. If a CCONTRL is not found and the connection limit is not exceeded, a CCONTRL is built. Once the CCONTRL is found or built, the CONN-OPEN REPLY buffer is chained to it.

After all PCT's have been processed in this manner, control proceeds to the next functional operation.

6.4.4.3. Receive Request

This functional operation is quite small, simply checking for new "receive" requests in all PCT's. In each PCT, the request-in-progress bit PCTFnRBZ is checked in each receive-request channel (n= 2, 3, and 4). If the bit is on, nothing further needs to be done; if the bit is off, the corresponding EXCH ECB is tested for completion.

If an EXCH completed abnormally, PCTFCLSE is turned on to terminate the process. When the EXCH completes normally, the channel is marked with request-in-progress by turning on PCTFnRBZ and the time-out interval received in the EXCH is changed to a time-of-day.

The same code is used to handle channel 0, although in this case the EXCH completion is the acknowledgment of an earlier reply to PL/MSG rather than a receive request.

When all PCT's have been processed, this functional operation is complete.

6.4.4.4. Connection Control

This functional operation, direct connection control, processes all CCONTRL blocks. For each CCONTRL chained from each PCT, the following connection control operations are performed:

* CCONDT is checked. If the timer has expired, flags are set to force M*S*G to either send CONN-CLOSE or assume that the remote CONN-CLOSE has been received.

* If the control sub-ptask has terminated, flags are set to send CONN-CLOSE and to rebuild the internal ECB wait list.

* If it is necessary to send CONN-CLOSE, a REPLY is obtained, the CONN-CLOSE item is created in it, and the REPLY is chained to PCTINQ for later transmission. CCONDT is set to time-out in five minutes, allowing the remote M*S*G this much time to respond with CONN-CLOSE.

* If CONN-CLOSE has been sent or CONN-REJ received and the sub-ptask is active, it is terminated with PDETATCH and then the flag is set to force rebuilding of the internal ECB wait list.

* If everything except notifying the local process has been done, the CCONTRL entry is dechained and transformed into a REPLY indicating connection termination. This REPLY is chained to PCTOUTQ.

* If both CONN-OPEN's have been received, MSGMAIN issues PPOST USER to wakeup the connection control sub-ptask, to allow it to continue processing.

* If the sub-ptask has supplied the local socket number, it is moved into the CONN-OPEN message, which is then chained to PCTINQ for transmission.

* The connection control sub-ptask is terminated with PDETACH if it still exists.

Each of the above steps is controlled by a flag bit in CCONTRL. When all CCONTRL entries have been processed, the next functional operation is started.

6.4.4.5. Route Created Events

This functional operation routes newly-created events, i.e., entries in each PCT's PCTINQ or SEQSTRM queue.

For each PCT, if the number of pending MESS requests has exceeded the quench configuration parameter N1, PCTFQNCH is set to indicate that the process should be quenched.

Each item in the SEQSTRM queues that is not delayed by "special handling" flags is moved (i.e., rechained) to the PCTINQ.

Every item in the PCTINQ that does not have an expired timer is moved either to the MSGCCNQ (if the destination process is local to this M*S*G) or to an HCTSENDQ for ARPANET transmission to a remote MSG. An ALARM item is moved to the beginning of the HCTSENDQ.

If the item is for a remote MSG and no HCT exists for that host, MSGMAIN requests that the LOGGER begin ICP to that host, and the item is moved to the MSGHCTQ to wait for a HCT. Only one ICP request is made for each unique host in the MSGHCTQ. A five minute time-out REPLY is created and chained to MSGSHPDT to abort MSGHCTQ entries, by host, if the remote host fails to respond.

An HCT with either the HCTFDUPE or the HCTFCLSE flag set is not valid for data transmission and will be skipped.

After all items that can be moved to the next processing state have been rechained in their new queues, the next functional operation can be performed.

6.4.4.6. Check for Created HCT

This functional operation is the smallest in MSGMAIN. Each item in the MSGHCTQ is compared against every HCT. If an HCT has been created by the out-going LOGGER request, the MSGHCTQ entries are moved to the HCTSENDQ and the MSGSHPDT timer entry is located and deleted.

6.4.4.7. HCT Processing

This functional operational, processing the Host Control Table entries to perform ARPANET input and output, is one of the largest in M*S*G.

First, output processing is done on each WRE. If the previous item has not been completely sent, nothing else can be done. Otherwise, the REPLY is either deleted or (for MESS or ALARM) moved to the HCTSENTQ to await an acknowledgment. The first entry in the HCTSENDQ, if any, is then moved to a queue which is "nidden" (from Rescind and time-out search), and ASEND is issued to send the item to the remote MSG.

If the operator has requested on-line debugging information, via the ACE, MSGMAIN issues a WTO for every item that is either sent or received. (Note that on-line debugging is per HCT and is not available if both processes are local.)

Input processing is much more complicated. Before M*S*G can do any processing of an incoming item, it must wait for the NCP to deliver the first two bytes of the item into the circular buffer in the HCT. These two bytes contain the length of the MSG item. Knowing this length, MSGMAIN obtains a REPLY buffer into which the entire item is copied from the circular buffer. When the NCP delivers an ARPANET message to the circular buffer and sends a "PPOST INPUT" wakeup signal to MSGAIN, MSGMAIN copies the bits into the REPLY buffer and then calls ARLSE to free the space in the circular buffer.

When the entire item has been assembled in the REPLY buffer, a flag is set in the HCT to indicate MSGMAIN must wait for the length bytes. The syntax of the item is checked, and a PROTOCOL ERROR is chained to the HCTSENDQ if the MSG protocol type is unknown. When any input is received, the HCTTIME timer, which controls idle HCT deletion, is reset to a new interval.

Depending on the item type, the following occurs:

* For a NOP, the REPLY buffer is deleted.

* An ECHO is changed to an ECHO-REP and chained to the HCTSENDQ.

* An ECHO-REP is deleted after an error WTO because CCN's MSG does not send ECHO.

* For a SYNCH, the version is checked and a PROTOCOL ERROR returned if the version is unsupported. A valid SYNCH sent in response to CCN's SYNCH is deleted; otherwise, it is changed into CCN's SYNCH response and chained to the HCTSENDQ.

* Any status or reserved protocol codes get PROTOCOL ERROR if an acknowledgment is needed or deleted with an appropriate WTO.

* A CLOSE sets HCT flags for later processing, and MSGMAIN closes the ARPANET receive socket.

* A MESS item is chained to the MSGCCNQ.

* MESS-REJ and MESS-OK cause a search of the HCTSENTQ for the MESS that is being acknowledged. The MESS item is deleted and the MESS-REJ or MESS-OK is transformed and chained to the MESS-sending process's PCTOUTQ to complete the pending event. Special handling status is updated, if necessary, or "out-of-sequence" is set. A quenched process is revived if the size of the pending-event set has decreased sufficiently.

* A MESS-HOLD is changed into a HOLD-OK and chained on the HCTSENDQ after the MESS is moved from the HCTSENTQ to the HCTHOLDQ. If no MESS is found, the MESS-HOLD is deleted and a longer MESS-CANCEL REPLY buffer is obtained and chained to the HCTSENDQ.

* A HOLD-OK is deleted without any processing.

* A MESS-CANCEL causes the XMIT REPLY in the PCTXMITQ (that would have requested the MESS) to be deleted along with the MESS-CANCEL REPLY buffer.

* XMIT is deleted after the corresponding MESS REPLY buffer is moved from the HCTHOLDQ to the HCTSENDQ. If no MESS is found, a MESS-CANCEL is obtained and chained to the HCTSENDQ, and the XMIT is deleted.

* ALARM, ALARM-OK, and ALARM-REJ are processed in a similar manner to MESS, MESS-OK, and MESS-REJ.

* CONN-OPEN, CONN-CLOSE, and CONN-REJ are chained to MSGCCNQ for later processing.

After each HCT in turn has been processed, the next functional operation is performed.

6.4.4.8. Check for Created PCT

This functional operation is the second smallest in MSGMAIN. Each item in the MSGPCTQ, representing a SendGeneric MESS, is compared with each PCT to see if it can be delivered (i.e., if there is an appropriate process with a pending ReceiveGenericMessage request). A deliverable item is moved to the beginning of the MSGCCNQ and the timer entry in MSGSHPDT is deleted. Only one item, at most, is moved to MSGCCNQ per cycle.

6.4.4.9. Route the MSGCCNQ

This functional operation, the largest and most complex one, routes all MSG items placed in the local-delivery queue MSGCCNQ by earlier functional operations. It is repeated until the MSGCCNQ is empty.

The first item in the queue is selected and the PCT's are searched for a process to receive it (i.e., for a process with the same generic name). If a process is found that has a ReceiveGenericMessage pending and the item is a SendGeneric MESS, a REPLY buffer is obtained and added to PCTRGMQ. The MESS is copied and expanded into this new REPLY buffer. The original REPLY buffer is either transformed into a MESS-OK and moved to the HCTSENDQ for return to a remote sending process, or it is transformed into a PL/MSG acknowledgment for a local process and moved to the sending PCT's PCTOUTQ.

If the item is not a SendGenericMessage, the candidate must be the specific process. When the correct PCT is found, the MESS or ALARM is processed like a SendGeneric MESS, except that it is chained to the PCTRSMQ or PCTEAQ. If too many SendSpecific items are queued for delivery, a MESS-HOLD is returned and an XMIT REPLY is chained to PCTXMITQ for later transmission to retrieve the MESS.

A CONN-OPEN is handled in a manner similar to a local ConnOpen request. The destination (local) PCT has been found, so the CCONTRL entry is found or built and the CONN-OPEN pair, if now available, is checked. CONN-CLOSE or CONN-REJ cause fields in the CCONTRL to be changed.

If a generic or specific PCT is not found and the item is not a SendGeneric MESS willing to wait for a process, or if the PCT was found but some other error occurred, an appropriate rejection must be returned. If the REPLY buffer is large enough to hold the reject message item (e.g., CONN-CLOSE to CONN-REJ), the new message is built in the REPLY which is chained to the HCTSENDQ or PCTOUTQ. Otherwise, the REPLY buffer is deleted and a larger one is obtained.

A SendGeneric MESS without a matching process is moved to MSGPCTQ while waiting for a PCT. A time-out entry for the waiting MESS is added to MSGSHPDT to abort the item if a process is not materialized in 5 minutes. MSGMAIN then calls PATTACH to create a sub-ptask executing MSGSTRT, which will start the requested process if possible.

When each item in the MSGCCNQ has been processed, MSGMAIN can go on to its next functional operation.

6.4.4.10. Search for Expired Timers

This functional operation is conceptually trivial. All data queues are searched for items that have expired timers. If one is found, it is suitably transformed and moved to the PCTOUTQ for the local process. "Out-of-sequence" condition is set in the appropriate SEQSTRM if necessary. If receive requests have expired, a dummy entry is added to the PCTRGMQ, PCTRSMQ, or PCTEAQ to complete the request. After all queues have been searched, the MSGSHPDT entries are scanned to abort any requests that failed to have a PCT or HCT created for them in the alloted time.

To avoid unusual "out-of-sequence" extra overhead, the SEQSTRM queues are not searched for time-outs. When an expired SEQSTRM item is eventually moved to the PCTINQ, a special check in the "Route Created Events" functional operation will leave the item in the PCTINQ for time-out processing.

6.4.4.11. Search for SYNCH Rejects

After all acknowledgments (including those for time-outs) have been delivered to the appropriate PCTOUTQ, the special handling "out-of-sequence" condition may exist. If so, any items held in the SEQSTRM queue because of special handling can be aborted. This functional operation scans all the entries in the SEQSTRM queues and aborts those that can be aborted. This pass also deletes idle SEQSTRM control blocks whose "SEQSTRM interval" timers have expired.

6.4.4.12. Return Data to PL/MSG

The next functional operation is performed on all PCT's. For all Exchange channels except 1, if there is something to be given to PL/MSG on the channel and the channel is free, the EXCH request is issued. If the channel is busy, the EXCH ECB is checked for completion. If an EXCH completed abnormally, PCTFCLSE is set. When the channel freed by the EXCH completing

normally, the REPLY is deleted and the EXCH requesting the time-out interval is issued. The PCTFnRBZ and PCTFnMBZ flags are set and reset as necessary to control channel status.

When a ReceiveSpecificMessage request has completed, if the sending process is local and has no other queued SendSpecificMessages (in the PCTRSMQ of the receiving process), the sending process will be revived if it was quenched.

6.4.4.13. Free Closed PCT's

The next functional operation, performed on all PCT's, deletes every one in which PCTFCLSE is set.

If a PCT to be deleted has CCONTRL entries, each entry is aborted but further processing of the PCT is skipped until the direct connection control functional operation has deleted all CCONTRL entries. Similarly, if any XMIT items are in the PCTXMITQ queue, the XMIT code is changed to MESS-REJ and the REPLY is moved to the PCTINQ queue, but further processing of the PCT is skipped so that the MESS-REJ item can be sent to the remote MSG.

When the CCONTRL and XMIT lists are empty, the MSGECBL flag is set to force the rebuilding of the ECB list. The Exchange window to PL/MSG is closed with PEXCLOSE, all remaining queue entries are deleted, and the PCT is dechained and freed with PCORE.

6.4.4.14. Free Closed HCT's

The next functional operation, performed on all HCT's, deletes every HCT that has been marked for close because either an MSG CLOSE item was received, the ARPANET connection broke, or the idle time-out expired. A CLOSE item is sent to the remote MSG if possible, but the HCT is not deleted until either an acknowledging CLOSE item is received or until a 5 minute close time-out interval expires.

If any items are chained to the HCT when it is ready to be deleted, the items are moved to MSGHCTQ, a MSGSHPDT entry is created, and the LOGGER notified to start ICP. No new connection will be requested if items were not pending. When the HCT is marked HCTFCLSE, any items that would normally be added to the HCTSENDQ cause a new HCT to be requested.

After all items has been moved to the MSGHCTQ, the ARPANET connections are closed, the ACE is deleted, and the HCT is dechained and freed.

This completes the description of the functional operations of MSGMAIN.

6.4.5. MSGSRVR

MSGSRVR is the transient module that creates a Host Control Table entry (HCT) in response to an incoming ICP request from a remote MSG. MSGSRVR is executed by a sub-ptask of LOGGER (actually, of a Host Control sub-ptask of LOGGER).

MSGSRVR first calls PCORE to obtain core for the HCT and initializes it; then it calls PLOAD to load the MSGTABC table module. If either the core or the module is unavailable, MSGSRVR calls PEXIT after issuing an appropriate ICT error WTO.

The ACE is completed from the information supplied by MSGTABC. Then the ARPANET send and receive sockets are opened; if this fails, an ICT WTO error message is issued, and PEXIT is called. When the sockets are open, AALLOC is used to send an allocation for the circular buffer in the HCT, and the WRE's in the HCT are initialized.

At this point, the MSGTABS tables are loaded and the entry for the remote host is found. The entry contains the authentication socket for the remote M*S*G. If MSGTABS cannot be loaded or the entry cannot be found, socket 31 (decimal) is assumed to be the authentication socket. Once the socket is known, the MSGTABS module is deleted. If the socket is even, authentication is not performed.

Standard ICP is performed using the authentication socket. If any NCP errors are detected while opening the socket or during reception of the data transfer socket number, an ICT error WTO is issued and PEXIT is called. When the data socket is received, the authentication socket is closed and the data socket is opened. An NCP error in data socket opening or authentication data transfer is treated like an ICP open error.

The data to be received is any combination of socket lists or socket ranges. A list is defined as a word containing the number of entries in the list followed by that many words, each containing a socket number. A range is defined as a word containing the value 0 followed by two words, the first containing the low socket and the second containing the high socket in the range.

The authentication data is processed, word by word, testing for a match with the socket passed to MSGSRVR when it was entered. Any such match means that the remote MSG is authentic. If the socket closes without a match (i.e., no valid socket was given) then MSGSRVR will issue an ICT error WTO and call PEXIT.

If authentication did not fail, MSGSRVR scans the PCLIST entries searching for MSGMAIN's PTA. If the PTA is found, MSGSRVR adds the new HCT to the HCT list anchored in the MSGHCTA field. It also checks the list for another HCT for the same remote host, and if it finds one it flags the new HCT for deletion by turning on its HCTFCLSE and HCTFDUPE flags. In any case, MSGSRVR then signals MSGMAIN with PPOST USER and calls PEXIT.

If MSGMAIN is not found, MSGSRVR deletes the MSGTABC module from core, initializes the PTA for MSGCONT, alters the PCLIST to make itself a direct sub-ptask of LOGGER (so that MSGMAIN will not be terminated when the ARPANET connection to this remote host closes), and calls PXCTL to transfer control to MSGCONT, thus transforming itself into MSGMAIN.

6.4.6. MSGSTEL

MSGSTEL is the interface module that services Server TELNET direct connections. MSGCONN has initialized the connection before using PXCTL to transfer control to MSGSTEL.

MSGSTEL first initializes variables that are location dependent. Then, to get things going, the EXCH requesting data from the process is issued. The ACE is checked for operator STOP or CANCEL bits; either causes the module to call PEXIT. If neither is set, the main PWAIT call is issued.

Control is returned to MSGSTEL when ARPANET input or output has completed, when a socket has closed, when an EXCH has completed, or when twenty minutes have elapsed. If control is returned because twenty minutes elapsed, PEXIT is called.

Each call to the TELNET routines passes the address of an Attention exit routine that only sets a flag indicating that an Attention has arrived. If this flag is now set, the flag is reset and an EXCH is issued on channel 0 if the channel is not busy. If the previous channel-0 EXCH is not completed, however, a previous Attention is still being transmitted, and the new one is discarded.

The next step is to determine the status of the data transfer from the NCP. If MSGSTEL is waiting for data from the NCP, ATGET is called to accept the available bytes. If none are available, the rest of the "get" operations are skipped. Any bytes returned by ATGET are passed to the process by a channel 1 EXCH. If MSGSTEL was not waiting for NCP data, it is waiting for the EXCH to complete. If the EXCH is not complete, the "put" status is checked. If the EXCH or ATGET complete abnormally, PEXIT is issued. When the EXCH completes, the next pass will call ATGET.

If the EXCH for data from the process is not yet completed, the ACE is checked for operator intervention, and the main PWAIT is issued. When the EXCH is complete, all available bytes will be given to the NCP by calling ATPUT. ATPUT will return control when the data has been transmitted. After ATPUT returns, an EXCH for more data is issued (i.e., the same EXCH that got things going in the beginning). If the EXCH or ATPUT terminates abnormally, MSGSTEL calls PEXIT.

6.4.7. MSGSTRT

MSGSTRT is a transient module which is executed by a sub-ptask of MSGMAIN and creates a local process when a SendGenericMessage arrives. MSTSTRT is given the address of the MESS REPLY containing the SendGeneric message. It first loads MSGTABC. If any error which prevents starting the process is detected, MSGSTRT sets an MSG reason code in the MESS REPLY and alters the MSGSHPDT entry for the MESS so that the MESS-REJ will be returned the next time MSGMAIN is given control.

The MSGTABC Process Start Table is scanned for the entry that represents the generic class. The entry specifies how to start the process (i.e., batch or TSO). A batch process is started by calling SRS to submit the JCL for the job.

For a TSO process, a TSO command is created in a core page obtained with a PCORE call. Next, MSGSTRT searches the PCLIST to find the PTA for MSGTSOC. When MSGTSOC is found, the command buffer is chained to PTAPARM, PPOST USER is issued, and MSGSTRT calls PEXIT. If MSGTSOC is not found, the buffer is chained to MSGSTRT's PTAPARM and MSGSTRT uses PXCTL to become MSGTSOC.

6.4.8. MSGTCAM

MSGTCAM is the interface module that services "encapsulated" TELNET direct connections. MSGCONN initializes the connection before using PXCTL to transfer control to MSGTCAM.

MSGTCAM first initializes variables that are location dependent and sets the connection state to output (i.e., waiting for output from TCAM) with an indicator to "flush" (i.e., not transmit) the output. This is done by convention to guarantee to the process that nothing is transmitted before the process issues a TGET.

MSGTCAM then issues its main PWAIT, waiting for ARPANET connection closed, EXCH request completed, or thirty minutes elapsed. The PWAIT does not wait for NCP input or output because these are handled by the TELNET routines. When control is returned, if the socket has closed or the time expired, MSGTCAM calls PEXIT. Because the only other event that could complete is EXCH, if the EXCH completed abnormally, MSGTCAM calls PEXIT. The ACE is checked and if the operator issued STOP or CANCEL, MSGTCAM also calls PEXIT.

If the on-line debugging flag is set in the PTA, the input and output lines are displayed by WTO.

Because an EXCH to TCAM has just completed, either the buffer that contained the data to be given to TCAM is free (because the virtual terminal is in input state) or TCAM has just presented output data to be transmitted to the user. The state flag is checked to determine which situation exists.

For output state, the data just passed from TCAM is given to the NCP using ATPUT, unless the data is to be flushed. After the ATPUT, an EXCH is issued for more data if the last data character was not X'FF', indicating TCAM has switched to input state. If the terminal switched states, processing after the ATPUT is as if the EXCH had completed in input state.

When the EXCH completes in input state, ATGET is called to request data from the NCP. The ATGET call indicates that control should not be returned to MSGTCAM until either the buffer is full or a complete line (i.e., string ending with ~NL) is received.

If control is returned from ATGET because the buffer is full, the input state is retained and the data is sent to TCAM without a trailing X'FF'. If the data received from the NCP ends with ~NL, an X'FF' is appended after the ~NL, the state is changed to output, and a conversational EXCH is issued with an MSG= buffer to pass the input and REP= to receive the output. After the EXCH to TCAM has been issued, the main PWAIT is executed.

Each call to the TELNET routines passes the address of an Attention exit routine that is to be given control if an Attention arrives. If the Attention arrives while in input state, an X'FF' (alone) is placed in the input buffer, and the conversational EXCH is issued after setting the state to output. If the Attention arrives in output state, MSGTCAM issues an EXCH reques on the other EXCH channel. When this EXCH completes, the EXCH for data from TCAM is issued after marking the state as output.

6.4.9. MSGTSOC

MSGTSOC is the TSO process controller module that manages the pool of TSO sessions. It is first entered by a PATTACH issued by the main M*S*G ptask, but later moves itself up in the ptask tree to be parallel to the main ptask.

MSGTSOC first loads MSGTABC, copies the session limits, and deletes the module. MSGTSOC then calls PCORE to obtain the control block table it uses for individual session control. There is one entry for each possible session, and each entry contains the session state, address of the TSO command buffer, address of the CCNUID, MSGTSOS PTA address, and MSGTSOS PTA termination internal ECB. Adjacent to the control blocks is the ECB list addressing each termination ECB.

MSGTSOC then verifies the PCLIST structure and alters it. When it is first entered, MSGTSOC is executing as a sub-ptask of MSGMAIN. After the alteration, the MSGTSOC ptask is a direct sub-ptask of LOGGER, parallel to MSGMAIN. Also, MSGMAIN is given a termination ECB. These changes are to notify MSGTSOC if MSGMAIN exits, allowing MSGTSOC to clean up.

Finally, MSGTSOC starts any sessions requested by the initial sessions parameter and then issues its main PWAIT. This request waits for a PPOST USER, for any sub-ptask to complete, or for thirty seconds. When the wait is completed, MSGTSOC searches its work queue. The number of TSO command buffers to be given to idle sessions are counted, deleting any that have expired timers. This count is added to the minimum idle sessions limit to get the number of idle sessions needed.

After the count of needed-sessions is obtained, the environment is checked. If the operator has set "no pseudo-users" or if MSGMAIN has exited, all sessions are terminated. If MSGMAIN has terminated, MSGTSOC then calls PEXIT itself. If the operator has set no pseudo-users, all further processing is bypassed (i.e., the main PWAIT is issued). Thus, MSGTSOC will check the

"no pseudo-users" bit every thirty seconds looking for a change.

If no sessions need to be started, this next phase is skipped. Each session control block is checked in turn. If the session has terminated, the control block is reinitialized by terminating the MSGTSOS ptask, clearing the PTA address and termination ECB, and deleting the CCNUID and MEMO. If the session is null or active, it is ignored. If the session is in LOGON, the count of needed sessions is decremented and the session is ignored. Any other session can only be idle. An idle session is given the first TSO command buffer, if any, and its control ptask (MSGTSOS) is awakened with PPOST USER. The count of needed sessions is then decremented, and if it is not zero the next session is examined.

When the count of needed sessions reaches zero, no new sessions are needed. However, if the search loop just described leaves a non-zero count of needed idle sessions, idle sessions are still needed. The maximum idle session limit added to the remaining count and the minimum idle session limit is subtracted, yielding the new number of sessions needed. The session control blocks are again searched and for each null entry a new MSGTSOS sub-task is created with PATTACH, until the count of needed sessions is exhausted. The session state of each such new session is set to LOGON and the PTA address is saved in the session control block. After starting the new sessions, MSGTSOC returns to the main PWAIT.

If it was not necessary to start any new sessions, more idle sessions than are permitted by the maximum idle session limit may exist. The session control blocks are scanned and any that represent terminated sessions are reinitialized. As the blocks are searched, a count of idle sessions is incremented. When the count exceeds the maximum idle limit number, any further idle sessions found are forcibly reinitialized. After all session control blocks have been scanned, the main PWAIT is again issued.

6.4.10. MSGTSOS

MSGTSOS is the TSO session control module. It is created as a sub-ptask by MSGTSOC to manage the virtual terminal controlling a single TSO pool session.

MSGTSOS first obtains a work area and then tries to establish an Exchange connection to TCAM. If the connection cannot be opened in thirty seconds, MSGTSOS issues an appropriate MSG error ICT WTO and calls PEXIT. Once the window is opened, MSGTSOS loads the MSGTABC module to obtain the CCN charge number and also selects a free pool Userid from the JOBNAME table. MSGTSOS then

deletes MSGTABC.

A CCNUID and MEMO are then built, identifying the LOGON directory. A TSO LOGON command is then created in the command buffer, and control passes from initialization into the main logic path as if MSGTSOC had just given a command buffer to MSGTSOS.

The command to be given to TCAM has ^NL and X'FF' appended to it. The EXCH that passes the command to TSO by the MSG= parm also waits for the output from the command by specifing REP=. After issuing the EXCH, MSGTSOS sets a flag to indicate the channel is in output state and then issues its main PWAIT. When the EXCH completes or thirty minutes elapse, MSGTSOS is again given control. If the EXCH completed abnormally or timed-out, MSGTSOS calls PEXIT.

All output data is discarded after ensuring that it does not indicate abnormal session completion. If the session completed, MSGTSOS calls PEXIT. If the output data does not end with X'FF' (indicating input state), a new EXCH with only REP= is issued to await more output data, and the main PWAIT is reissued.

If the channel has gone to input state, the MSG process is assumed to have completed. The session control block is examined for session state and the following action taken as appropriate:

* If the session has just finished LOGON, the session state is changed to idle. MSGTSOC is awakened with PPOST USER, and MSGTSOS then itself issues PWAIT USER. When a command is available, MSGTSOC will PPOST MSGTSOS, having placed the address of the command in the session control block. MSGTSOS copies the command, frees the core, and changes the session state to active. The copied command is then given to TCAM as the LOGON command was.

* If the session has just finished being active, the session state is changed to idle and a FREEALL command is built and sent to TCAM.

* If the session has just finished being idle, the FREEALL has just completed. As is done after LOGON completes, MSGTSOS issues a PPOST to awaken MSGTSOC and then issues PWAIT USER itself to await the next command.

MSGTSOS continues to wait for work and process commands, discarding the output data, until either it issues PEXIT, or MSGTSOC terminates it as an excess idle session.

If the on-line debugging flag is set in the PTA, the input and output lines are displayed by WTO.

6.4.11. MSGUICP

MSGUICP is a transient module which creates a Host Control Table entry (HCT) for an outgoing ICP request. MSGUICP is executed by a sub-ptask of LOGGER (actually, of an NCP Host Control sub-ptask of the LOGGER ptask). MSGUICP operates almost exactly like MSGSRVR.

MSGUICP first uses PCORE to obtain the HCT and initializes it, and then uses PLOAD to load the MSGTABC table module. If either the core or MSGTABC is unavailable, MSGUICP issues an appropriate ICT error WTO and then calls PEXIT.

The ACE is completed from the information supplied by MSGTABC. Then the ARPANET input and output sockets are opened. A PEXIT after an ICT error WTO is issued if the sockets fail to open properly. When the ARPANET connections are open, AALLOC is used to send an allocation for the circular buffer in the HCT; then the WRE's in the HCT are initialized.

MSGUICP next scans the PCLIST searching for MSGMAIN's PTA. If the PTA is found, MSGSRVR adds the HCT to MSGMAIN's MSGHCTA list and flags the HCT with HCTFCLSE and HCTFDUPE if another HCT exists to the same remote M*S*G. If the HCT is not a duplicate, MSGUICP obtains a REPLY buffer and builds an MSG SYNCH item to be sent to the remote M*S*G. MSGUICP then signals MSGMAIN with a PPOST USER call, and then calls PEXIT. If the MSGMAIN PTA is not found, MSGUICP issues an ICT error WTO and then calls PEXIT.

6.4.12. MSGUSER

The MSGUSER module is executed by a sub-ptask of the LOGGER ptask, created when a local process issues an EXOPEN in order to materialize itself as an MSG process. MSGUSER must verify that the job has access to MSG and, if so, build a Process Control Table entry (PCT) for this local process.

MSGUSER calls PLOAD to load the MSGTABC module and scans the jobname entries to verify that the job is allowed MSG access. If the job is not allowed, MSGUSER issues an appropriate ICT error WTO and calls PEXIT.

If the job is valid, the tables are deleted from the region with PDELETE, and PCORE is called to obtain core for the PCT. An EXCH request is issued to obtain the generic name of the process from PL/MSG. If the EXCH

completes abnormally, PEXIT is called. Otherwise, the generic name and its length are saved in the PCT, and the name is forced to be uppercase EBCDIC. A zero-length generic name is invalid. The MSGTABS tables are loaded, searched for a generic code to be saved in the PCT, and then deleted.

The rest of the PCT EXCH control blocks are then initialized, and a 2K area for REPLY buffers is obtained in subpool 2 (or 3) with GETMAIN. The address of the 2K area is saved in the PCT.

The PCLIST entries are then searched for MSGMAIN's PTA. If the PTA is found, the process instance number is incremented and the complete process name is saved in the PCT. The PCT is then chained to the MSGMAIN work queues and a wakeup call to MSGMAIN is sent with PPOST USER. Finally, the MSG host number, incarnation, and process instance are passed to PL/MSG and therefore the proess via the Exchange window, and then MSGUSER calls PEXIT.

If the MSGMAIN PTA is not found, MSGUSER converts itself to MSGMAIN in the following manner. It initializes its PTA for MSGMAIN and generates a new incarnation number. The process instance number is set to one and the complete process name is saved in the PCT, which is chained to MSGPCTA as the only PCT. The MSG host number, incarnation, and process instance are passed to PL/MSG as before, after which MSGUSER calls PXCTL to transfer control to MSGCONT and complete MSGMAIN initialization.

6.4.13. MSGUTEL

MSGUTEL is the interface module that services User TELNET direct connections. MSGCONN has initialized the connection before transferring control to MSGUTEL. MSGUTEL is almost identical to MSGSTEL in operation.

MSGUTEL first initializes variables that are location dependent and issues the EXCH request on the Attention channel. Then an EXCH is issued requesting data from the process. Next, the ACE is checked for operator STOP or CANCEL; either causes a call of PEXIT. If neither has been set by the operator, MSGUTEL issues the main PWAIT. The main difference between MSGSTEL and MSGUTEL is in Attention processing.

Control is returned to MSGUTEL when input or output to the NCP is complete, when a socket has closed, when an EXCH has completed, or when twenty minutes have elapsed. If control is returned because twenty minutes elapsed, PEXIT is called.

MSGUTEL always has a pending EXCH that uses an interrupt signal on the Attention channel. When that EXCH completes (indicating that an Attention must be sent), an asynchronous routine is given control to set a flag and mark MSGUTEL ready. When execution continues after PWAIT, the flag is tested; if it is set, it is reset and the NCP routine AINT is called to send the Attention.

The next step is to determine the status of the data transfer from the NCP. If MSGUTEL is waiting for data from the NCP, ATGET is called to accept the bytes available or to discover that nothing is available -- in which case, the rest of the "get" operations are skipped. Any bytes returned by ATGET are given to the process by a channel 1 EXCH. When MSGUTEL is not waiting for data from the ARPANET, it is waiting for the NCP-to-process EXCH to complete. If the EXCH has not completed, the "put" status is checked. If the EXCH or ATGET complete abnormally, PEXIT is called. When the EXCH completes, the next pass will call ATGET.

The "put" operations are slightly different for MSGUTEL. First, the status of the ATPUT is checked and if the previous ATPUT has not completed, ATPUT is again called to continue with the previous buffer. If the ATPUT was complete, the process-to-NCP data EXCH is tested for completion. If this EXCH is not yet completed, the ACE is checked for operator intervention and the main PWAIT is issued. If the EXCH is complete, the available bytes are sent to the ARPANET by calling ATPUT. When ATPUT returns, if the buffer has been completely transmitted, the EXCH for more data is issued (i.e., the same EXCH that got things going in the beginning). If the ATPUT is not completed, the EXCH is skipped. In either case, the ACE is checked and the main PWAIT is issued. If either the EXCH or ATPUT terminate abnormally, PEXIT is called.

6.4.14. MSGVRFY

MSGVRFY is a transient module that services an incoming authentication request. MSGVRFY is executed as a sub-task of LOGGER (actually, of the Host Control ptask which is a sub-ptask of LOGGER).

MSGVRFY issues PCORE to obtain a work area and then issues PLOAD to load the MSGTABC table module. If either the core or the module is unavailable, MSGVRFY issues an appropriate ICT error WTO and calls PEXIT.

The ACE is completed from the information supplied by MSGTABC, and the ARPANET send socket is opened. If the connection fails to open properly, than an ICT error WTO is issued and PEXIT is called.

After the socket is open, the PCLIST is scanned to find MSGMAIN's PTA. The MSGHCTA list (i.e., the Host Control Table) is scanned and a socket list containing the HCT sockets that are the CCN ends of open inter-MSG connections is sent to the host requesting authentication. See the MSGSRVR description for the format of this data. After the socket list has been sent, MSGVRFY calls PEXIT.

If the MSGMAIN PTA is not found or if no connections are open to the host requesting authentication, MSGVRFY issues an ICT error WTO before calling PEXIT.

APPENDIX A :: PL/MSG EXCEPTIONAL CONDITIONS

The PL/MSG exceptional conditions of interest correspond to the names of the values returned in event signals. These values are defined by %INCLUDE packet MSGDISP. The meanings of these names are listed here.

* SUCCESSFUL -- the event completed normally.

* ALREADY_PENDING -- the set of pending events already includes as many events of this class as is permitted.

* BAD_LENGTH_BUFFER -- A message area which is to receive a MSG message has a maximum length field that is either zero or greater than the implementation permits.

* BAD_BYTE_SIZE -- a connection byte size is not in the range 1-255.

* BAD_CONN_TYPE -- a connection type code is unknown.

* MESSAGE_TOO_LONG -- An incoming message was longer than the maximum length that the caller was willing to accept. The message has been lost.

* UNKNOWN_OPTION_BITS -- This code cannot occur with the current version of PL/MSG.

* RESCINDED -- the pending event was aborted in response to a rescind primitive.

* NAME_TABLE_LOAD_ERROR -- an error has occurred within the internal management of the MSG task. If this occurs, MSG needs maintenance.

* BAD_HOST -- the host number field of a process name contains a number that is not assigned to an NSW host computer.

* BAD_INCARNATION -- the MSG of the host being addressed has been reincarnated, and the requested process no longer exists under the given name.

* BAD_INSTANCE -- the addressed instance of the process no longer exists.

* BAD_GENERIC_LENGTH -- the length of a generic name string is not in the range 1-127.

* BAD_GENERIC_NAME -- the generic name string is not recognized by the addressed host.

* MUST_RESYNC -- transmission of this type of message to this process is inhibited due to a failure to deliver a sequenced or stream-marked message. The operation might be successful if retried after an appropriate resynch primitive is issued.

* UNKNOWN_PRIMITIVE -- there has been a communication breakdown between PL/MSG and the network MSG task. This probably means either that you are using an out-of-date PL/MSG package or that your program has destroyed PL/MSG storage.

* UNKNOWN_GENERIC_CODE -- the process class of a SendGenericMessage is unknown at the requested host.

* ALARM_NOT_ACCEPTED -- the process to which an alarm was sent is not armed for alarms.

* ALARM_QUEUED -- the process to which an alarm was sent does not have an enabled ReceiveAlarm primitive pending, but is armed for alarms. The alarm was queued.

* UNKNOWN_DESTINATION -- the host number field of a process name contains a number that is not assigned to an NSW host computer.

* NO_RECEIVE_PENDING -- a SendGenericMessage cannot be associated with a ReceiveGenericMessage without being queued. Because the caller has requested that the message not be queued, the message has been rejected.

* TIMEOUT_EXCEEDED -- a primitive failed to complete within the time specified as its time-out value, and it has been aborted.

* DESTINATION_QUEUE_FULL -- the intended recipient of an outgoing MSG message is backlogged and is rejecting messages.

* MY_PROCESS_NAME_BAD -- an attempt has been made to materialize a local process with an invalid generic name.

* NO_RESOURCES_TO_START -- a process must be started to satisfy a SendGenericMessage, but the host is saturated in one way or another, and new processes cannot be started just now.

* UNKNOWN_ERROR -- what you were asking for didn't work, but no one can say just why.

* DESTINATION_HOST_DIED -- the addressed process resides on a host which is reported by the network to be down.

* DUPLICATE_CONNECTION -- an OpenConn has been requested for a remote process and connection id for which a connection already exists. CloseConn has not been completed for the old connection.

* NO_RESOURCES_FOR_CONN -- not all the mechanisms needed to satisfy an OpenConn are presently available.

* PROCESS_REQUESTED_CLOSE -- a direct connection has been closed normally by one or the other of the using processes.

* CONNECTION_ABORTED -- a direct connection has been closed abnormally due to a failure in the network communication mechanisms that support it.

* NO_SUCH_CONNECTION -- an operation, specifying a direct connection which does not exist, has been requested.

* NO_SUCH_PROCESS -- an operation, specifying a process name that does not stand for a known process, has been requested.

* BAD_CONNECTION_ID -- a direct connection cannot be opened because the remote process is requesting a different connection identifier.

* TYPE_MISMATCH -- a direct connection cannot be opened because the remote process is requesting an incompatible connection type code.

* BYTE_MISMATCH -- a direct connection cannot be opened because the remote process is requesting a different connection byte size.

* CANNOT_OPEN_CONNECTION a direct connection cannot be opened because of any other reason.

* TRANSMISSION_ERROR -- a connection has been closed in such a way that PL/MSG cannot guarantee that all transmitted data was delivered first.

APPENDIX B -- M*S*G ERRORS AND REMEDIES

Each error situation that follows has three items of information:

* The name of the module issuing the M*S*G error ICT SVC.

* The text of the error WTO message.

* The error description and suggested remedial action.

The parameter passed to the M*S*G error ICT SVC to select a particular message text is the number that matches 'nnn' in the text header, MSGnnnx. If the 'x' in the header is 'I', the message is informational; if the 'x' is 'A', some action is required of the central operator.

MSGMAIN
MSG000I (HOST TO HOST DEBUGGING INFO)
IF EXCESSIVE NUMBER OF WTO'S, CONTACT THE NSW PROGRAMMER WHO TURNED ON WTO.

MSGTSOS
MSG001I (LOCAL PROCESS DEBUGGING INFO)
IF EXCESSIVE NUMBER OF WTO'S, CONTACT THE NSW PROGRAMMER WHO TURNED ON WTO. ALSO CAN BE GENERATED FROM MODULE MSGTCAM BY ENCAPSULATION DIRECT CONNECTION.

IGC65ICT
MSG002A UNKNOWN MSG ERROR
ERROR WTO ICT SVC CALLED WITH INVALID ARGUMENT. CONTACT THE NSW PROGRAMMER.

MSGTSOC
MSG003A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGTSOC
MSG004A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
ARPA REGION PROBABLY FULL.

MSGTSOC
MSG005A PCLIST ERROR
INTERNAL MSG TASK STRUCTURE ERROR. TAKE SNAP DUMP OF ARPA.
CONTACT THE NSW PROGRAMMER.

MSGTSOC
MSG006I TSO NOT AVAILABLE
REQUEST FOR PROCESS TIMED-OUT BECAUSE: 1) TSO NOT UP 2) TSO IN NO-PSEUDO 3) TSO TOO SLOW. CONTACT THE NSW ADMINISTRATOR.

MSGTSOS
MSG007A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGTSOS
MSG008A TCAM NOT AVAILABLE
EXOPEN FOR VIRTUAL PORT TIMED-OUT. PROBABLY NO PORTS AVAILABLE. CONTACT THE NSW ADMINISTRATOR.

MSGTSOS
MSG009A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGTSOS
MSG010I LOGON USERID NOT AVAILABLE
ALL NSW TSO USERID'S IN USE. CONTACT THE NSW ADMINISTRATOR.

MSGTSOS
MSG011A UID CREATE FAILED
UID CREATE SSVC RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.

MSGTSOS
MSG012I PROCESS PRESUMED DEAD
TIME-OUT WAITING FOR TPUT/TGET. CONTACT THE NSW PROGRAMMER.

MSGSTRT
MSG013A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGSTRT
MSG014I UNKNOWN GENERIC CLASS
GENERIC NAME NOT IN MSGTSOC. PROBABLY SOMEONE TESTING. CONTACT THE NSW ADMINISTRATOR.

MSGSTRT
MSG015A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
ARPA REGION PROBABLY FULL.

MSGSTRT
MSG016A SRS FAILED
SRS RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
PROBABLY MEMBER OR SUBMIT DATASET MISSING.

MSGSTRT
MSG017A UNKNOWN START TYPE
MODULE MSGTABC HAS AN INCORRECT ENTRY. CONTACT THE NSW
PROGRAMMER.

MSGSTRT
MSG018A PCLIST ERROR
INTERNAL MSG TASK STRUCTURE ERROR. TAKE SNAP DUMP OF ARPA.
CONTACT THE NSW PROGRAMMER.

MSGSTRT
MSG019I MAXIMUM ACTIVE PROCESSES
REQUEST TO START PROCESS REJECTED BECAUSE MAXIMUM NUMBER OF
LIKE PROCESSES ACTIVE. CHECK MSGTABC FOR INCORRECT MAXIMUM.
CONTACT THE NSW PROGRAMMER AND ADMINISTRATOR.

MSGSTRT
MSG020A PXCTL FAILED
PXCTL RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
MODULE MSGTSOC PROBABLY SCRATCHED FROM LIBRARY OR NO CORE
FOR MODULE.

MSGINIT
MSG021A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
MODULE MSGTABS PROBABLY SCRATCHED FROM LIBRARY.

MSGINIT
MSG022A PXCTL FAILED
PXCTL RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.
MODULE MSGMAIN PROBABLY SCRATCHED FROM LIBRARY OR NO CORE
FOR MODULE.

MSGINIT
MSG023A GETMAIN FAILED
GETMAIN RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. PROBABLY NO CORE FOR MSG BUFFER SUBPOOL.

MSGSRVR
MSG024A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGSRVR
MSG025A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGSRVR
MSG026A ALSTN FAILED
ALSTN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM.

MSGSRVR
MSG027A AOPEN FAILED
AOPEN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGSRVR
MSG028A CCB CLOSE ERROR
CCB MARKED CLOSED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGSRVR
MSG029A CCB TIME-OUT
PWAIT RETURNED BECAUSE TIMER EXPIRED WHILE WAITING FOR AOPEN OR DATA. CONTACT THE NSW PROGRAMMER. INDICATES THAT THE REMOTE MSG IS NOT WORKING CORRECTLY.

MSGUSER
MSG030A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGUSER
MSG031I INVALID JOB ATTEMPTED MSG ACCESS
REQUEST FOR MSG SERVICE FROM UNAUTHORIZED JOB. CONTACT THE NSW ADMINISTRATOR.

MSGUSER
MSG032A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGUSER
MSG033A EXCH ERROR
EXCH RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER.

MSGUSER
MSG034A GETMAIN FAILED
GETMAIN RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. PROBABLY NO CORE FOR NEW BLOCK IN MSG BUFFER SUBPOOL.

MSGCONN
MSG035A AACEBUY FAILED
AACEBUY RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. PROBABLY NO CORE FOR ACE.

MSGCONN
MSG036A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGCONN
MSG037A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGCONN
MSG038A ALSTN FAILED
ALSTN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM.

MSGCONN
MSG039A AOPEN FAILED
AOPEN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGCONN
MSG040A CCB CLOSE ERROR
CCB MARKED CLOSED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGCONN
MSG041A ATOPEN FAILED
ATOPEN RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGCONN
MSG042I INVALID TCAM CONNECTION REQUEST
CONNOPEN REQUEST TYPE=TCAM COULD NOT BE SERVICED BECAUSE OF ONE OF THE FOLLOWING 1) PROCESS REQUESTING SERVICE NOT STARTED BY MSG IN TSO POOL 2) WORKAREA OR WINDOW FOR TCAM COULD NOT BE FOUND. IF PERSISTS, CONTACT THE NSW PROGRAMMER.

MSGUICP
MSG043A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGUICP
MSG044A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGUICP
MSG045A ALSTN FAILED
ALSTN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM.

MSGUICP
MSG046A AOPEN FAILED
AOPEN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGUICP
MSG047A CCB CLOSE ERROR
CCB MARKED CLOSED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGUICP
MSG048A CCB TIME-OUT
PWAIT RETURNED BECAUSE TIMER EXPIRED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES THAT THE REMOTE MSG IS NOT WORKING CORRECTLY.

MSGUICP
MSG049A MSGMAIN GONE
SCAN OF PCLIST ENTRIES FAILED TO FIND MSGMAIN(X). CONTACT THE NSW PROGRAMMER. INDICATES THAT MSGMAIN ABENDED OR THAT MSGUICP WAS INCORRECTLY GIVEN CONTROL.

MSGVRFY
MSG050A PCORE FAILED
PCORE RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. ARPA REGION PROBABLY FULL.

MSGVRFY
MSG051A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABC PROBABLY SCRATCHED FROM LIBRARY.

MSGVRFY
MSG052A ALSTN FAILED
ALSTN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM.

MSGVRFY
MSG053A AOPEN FAILED
AOPEN RETURN CODE GREATER THAN 4. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGVRFY
MSG054A CCB CLOSE ERROR
CCB MARKED CLOSED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES ARPA INTERNAL NCP PROBLEM OR POSSIBLY REMOTE HOST DIED.

MSGVRFY
MSG055A CCB TIME-OUT
PWAIT RETURNED BECAUSE TIMER EXPIRED WHILE WAITING FOR AOPEN. CONTACT THE NSW PROGRAMMER. INDICATES THAT THE REMOTE MSG IS NOT WORKING CORRECTLY.

MSGVRFY
MSG0056A MSGMAIN GONE
SCAN OF PCLIST ENTRIES FAILED TO FIND MSGMAIN(X). CONTACT THE NSW PROGRAMMER. INDICATES THAT MSGMAIN ABENDED OR THAT MSGVRFY WAS INCORRECTLY GIVEN CONTROL.

MSGVRFY
MSG0057I NO LINKS OPEN
REQUEST FOR VERIFICATION CAME FROM HOST TO WHICH MSG HAS NO OPEN COMMUNICATION PATH. CONTACT THE NSW ADMINISTRATOR AND PROGRAMMER. EITHER MSG IS NOT WORKING CORRECTLY OR SOMEONE IS TRYING TO BREAK MSG SECURITY.

MSGMAIN
MSG0058A GETMAIN FAILED
GETMAIN RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. PROBABLY NO CORE FOR NEW BLOCK IN MSG BUFFER SUBPOOL. ROUTE DUMP TO NSW PROGRAMMER.

MSGMAIN
MSG0059A PLOAD FAILED
PLOAD RETURN CODE NON-ZERO. CONTACT THE NSW PROGRAMMER. MODULE MSGTABS PROBABLY SCRATCHED FROM LIBRARY.

By using the operator interface to the NCP, a system programmer can turn on debugging WTO's for a particular inter-MSG connection. These WTO's have the following form:

MSG000I ttt ddd sss llllll ccc pppp

where:

* ttt describes the transaction:

  OUT -- Item being sent from CCN to remote MSG.

  INP -- Item received by CCN from remote MSG.

  BUG -- Logic error in CCN's M*S*G.

  ERM -- Remote MSG sent this PROTOCOL ERROR.

  DEL -- This item was deleted to avoid logic error.

  ERH -- CCN's M*S*G is sending this PROTOCOL ERROR.

  Note: OUT and IN options are normally disabled but can be enabled by the ECH option in the NETSTAT or AR processors. The others are always enabled.

* ddd is the MSG destination host number (decimal).

* sss is the MSG source host number (decimal).

* llllll is the length of the inter-MSG item (decimal).

* ccc is the type code (the third byte of an inter-MSG item) in decimal.

* ppp is the PROTOCOL ERROR code, if applicable (hex).

APPENDIX C -- M*S*G-TO-M*S*G REASON CODES

The following hexadecimal codes may be sent by CCN's M*S*G to a remote MSG. For each code, the type of inter-MSG item in which it may be sent and its meaning are shown. The following mnemonics are used for the type of inter-MSG items:

AR - ALARM-REJ
CC - CONN-CLOSE
CR - CONN-REJ
MC - MESS-CANCEL
MR - MESS-REJ
PE - PTCL-ERR

'C001' - PE - Protocol command not implemented.
'C002' - PE - Unknown protocol command.
'C003' - PE - Protocol command syntax error.
'C005' - PE - SYNCH version unsupported.
'C041' - AR/MR - Unknown destination process.
'C042' - MR - Process queue full.
'C045' - AR/MR - Destination host incarnation bad.
'C082' - MC - Messages rescinded or timed-out.
'C0C1' - MR - Insufficient resources to start process.
'C101' - AR - Process not accepting alarms.
'C102' - AR - Process has an alarm queued already.
'C141' - MR - Generic class not used here.
'C142' - MR - Can't start another like process.
'C143' - MR - Nowait specified and no process pending.
'C181' - CC - Process requested CONNCLOSE.
'C182' - CC - Responding to CONNCLOSE.
'C184' - CC - Connection TYPE unknown.
'C186' - CR - No such connection in CONNCLOSE.
'C187' - CR - No such process in CONNOPEN or CONNCLOSE.
'C188' - CR - Duplicate connection request invalid.
'C189' - CC - Connection TYPE mismatch.
'C18A' - CC - Connection aborted; other process had error.
'C18B' - CC - Byte-size mismatch.
'C18C' - CC - Process died.
'C18D' - CC - Time-out waiting for remote CONNOPEN.
'C18E' - CC - Local interface time-out error.
'C18F' - CC - No resources for connection.
'C190' - CC - Exceptional condition during data transfer.

The inter-M*S*G reason codes and the mnemonics used by CCN's M*S*G are as follows:

'C001' - HCNOTIMP
'C002' - HCUNKPRT
'C003' - HCSYNTAX
'C004' - HC2LONG - MSG message too long.
'C005' - HCVSYNCH
'C041' - HCUNKPRC
'C042' - HCPRCFUL
'C043' - HCGENSPC - Destination Generic/Specific mismatch.
'C044' - HCBADGEN - Generic name unknown.
'C045' - HCBADINC
'C081' - HCSRCNAM - Source process name bad.
'C082' - HCRSCTIM
'C0C1' - HCNORESR
'C101' - HCNOALRM
'C102' - HCQDALRM
'C141' - HCNOGEN
'C142' - HCCNTGEN
'C143' - HCWONTW8
'C181' - HCUSERCL
'C182' - HCACKNCL
'C183' - HCRECCLS - Received CLOSE for connection.
'C184' - HCBADCON
'C185' - HCCONPRC - Process name mismatch.
'C186' - HCNOCONN
'C187' - HCLCLPRC
'C188' - HCIDBAD
'C189' - HCBADTYP
'C18A' - HCCONABR
'C18B' - HCBADBYT
'C18C' - HCPRCDIE
'C18D' - HCPRCTIM
'C18E' - HCHSTTIM
'C18F' - HCNORESO
'C190' - HCXMITER

APPENDIX D -- M*S*G-TO-PL/MSG COMPLETION CODES

The following decimal ECB completion codes are generated in either PL/MSG or M*S*G to inform a local process of the normal or abnormal completion of the request:

0 - NORMAL - Request completed normally.
1 - PENDING - Request already pending.
2 - ZEROLEN - Zero length message.
3 - ZEROBITS - Zero byte-size.
4 - WHATTYPE - Unknown connection TYPE.
5 - LOSTDATA - Lost data (buffer too small).
6 - BADBITS - Invalid code bits set.
7 - RESINDED - Pending request rescinded.
8 - NONAMES - Internal M*S*G PLOAD error.
9 - BADHOST - Invalid host number.
10 - BADHOSTI - Invalid host incarnation.
11 - BADPRCSI - Invalid process instance.
12 - BADCOUNT - Invalid generic name length.
13 - BADGENER - Unknown generic name.
14 - RESYNCH - Must RESYNCH path.
15 - WHATPRIM - Unknown primitive code.
16 - BADSHORT - Unknown generic code.
17 - NOTACPT - Process not accepting alarms.
18 - ALARMQD - Process has alarm queued.
19 - BADPRCSS - Destination process unknown.
20 - CANTSTRT - Can't start another process.
21 - WONTWAIT - Won't wait for new process.
22 - TIMEDOUT - Pending request timed-out.
23 - DESTQFUL - Destination queue full.
24 - BADMYNAM - Local generic name bad.
25 - NORESRCS - No resources to start process.
26 - UNKNOWNC - Unknown MSG error code.
27 - DSTHSTDI - Destination host died.
28 - DUPCONN# - Duplicate connection id.
29 - NORESCON - No resources for connection.
30 - NORMCLSE - Connection closed normally.
31 - ABORCLSE - Connection aborted.
32 - NOSUCHCN - No such connection.
33 - NOSUCHPR - No such process.
34 - BADCONNI - Duplicate connection requested.
35 - BADTYPPR - Connection TYPE mismatch.
36 - BADBYTPR - Byte-size mismatch.
37 - CNTCONOP - Cannot AOPEN path.
38 - CONNXMIT - Transmission or EXCH error.

APPENDIX E -- MSG CONFIGURATION

MSGTABC MODULE ASSEMBLY

Figure 8 illustrates a sample assembly source for a MSGTABC configuration module. The module is structured with a fixed-format header followed by variable-length data and tables. The header, which includes pointers to the beginning and end of the various tables and other variable data items, has the following format (using the notation of System/360 Assembly language):

* A(M*S*G Directory)

  This is the address of a ten-byte character string that appears later in the variable portion of the module.

* A(first Valid Jobname Table entry)

* A(last Valid Jobname Table entry)

* A(first TSO pool entry in Valid Jobname Table)

  Note: the TSO pool entries must be collected at the end of the Valid Jobname Table.

* A(Process Charge Number)

  Address of a six-byte character string containing the CCN Charge Number to be used for processes started under TSO. This may be the same as the Charge Number in the M*S*G Directory parameter.

* H(Startup Session Count)

  The number of TSO pool sessions to be started automatically when M*S*G is restarted.

* H(Minimum Idle Session Count)

  The minimum number of idle pool sessions that M*S*G allows before it must start another idle session.

* H(Maximum Idle Session Count)

  The maximum number of idle pool sessions that M*S*G allows before it must delete excess idle sessions.

* H(Maximum Session Count)

  The total number of TSO pool sessions, both idle and active, that M*S*G allows at any one time.

* A(first Process Start Table entry)

* A(last Process Start Table entry)

Following this fixed-format header, the following variable data and tables can occur in any order:

* M*S*G Directory

  A ten-byte character string containing the Charge Number (six characters and a blank) and a Userid (three characters) to be used when writing accounting records from M*S*G ptasks within the NCP.

* Process Charge Number

  A six-character string. This can be omitted if the Charge Number is the same as the Charge Number part of M*S*G Directory.

* Valid Jobname Table

  A list of JOBNAME macros, with the TSO pool jobnames collected at the end. The first and last macros and the first TSO pool macro have labels that define the addresses in the fixed information at the beginning. See the example below.

* Process Start Table

  A series of PROCESS macros. Again, the first and last are pointed to by addresses in the header.

MSGTABS MODULE ASSEMBLY

Figure 9 illustrates a sample assembly source for the MSGTABS configuration module. This module contains the MSG-dependent information and is structured like the MSGTABC module, with a fixed-format header followed by set of variable tables. The header has the format:

* A(first Generic Class Table entry)

* A(last Generic Class Table entry)

* A(first MSG Host Table entry)

* A(last MSG Host Table entry)

* A(first Remote Address Table entry)

* A(last Remote Address Table entry)

* A(first Error Code Table entry)

* A(local M*S*G entry)

  The address of the entry in the MSG Host Table for the local M*S*G.

* F(SEQSTRM interval)

  The time interval, in .01 seconds, to keep an idle process-to-process path queue control block. To indicate no timing, use zero.

* F(HCT interval)

  The time interval to keep an idle inter-MSG connection before closing it.

* H(quench limit)

  The number N1, limiting the number of pending SendSpecifics for a local process.

* H(hold limit)

  The number N2, limiting the number of messages queued for delivery to a local process before M*S*G starts sending MESS-HOLD to remote M*S*G's.

* H(xmit limit)

The number N3, limiting the number of messages to a local process which remote MSG's can be asked to queued pending later XMIT's.

* H(connect limit)

  The maximum number of direct connections a process can have simultaneously.

The variable-length tables follow the header. Each table entry, except for the Error Code Table, is generated by macros; the tables may appear in any order.

* Generic Class Table

  This table is generated by CLASS macros. For convenience, the symbol 'CCN' is defined in M*S*G to be CCN's host number. See the example below.

* MSG Host Table

  This table is generated by HOST macros.

* Remote Address Table

  This table is generated with the macro REMOTE.

* Error Code Table

  As may be seen in the example below, it consists of a list of pairs of halfwords. Mnemonic symbols are explained in previous appendices. No macro exists for this table. The correspondence between inter-M*S*G reasons and PL/MSG codes should not be changed since the reasons are defined by MSG protocol and all existing (local) processes would be affected by changes in this table.

```
//NSWMSG  EXEC PGM=TSOTIME,PARM='#FIRST'
//STEPLIB DD DISP=SHR,DSN=ABC123.NSW.CMDLIB,
//           UNIT=2314,VOL=SER=NSWP01
//SYSPROC DD DISP=SHR,DSN=ABC123.NSW.CMDPROC,
//           UNIT=2314,VOL=SER=NSWP01
//         DD DISP=SHR,DSN=SYS1.CMDPROC,
//           UNIT=2301,VOL=SER=DRUM02
//DD1 DD DYNAM
//DD2 DD DYNAM
//DD3 DD DYNAM
//DD4 DD DYNAM
//DD5 DD DYNAM
//DD6 DD DYNAM
//DD7 DD DYNAM
//DD8 DD DYNAM
//DD9 DD DYNAM
//SYSUDUMP DD SYSOUT=A,SPACE=(TRK,(50,50),RLSE)
//SNAPDD DD SYSOUT=A,SPACE=(TRK,(20,20),RLSE),
//          DCB=BLKSIZE=882
```

Figure 7. Sample TSO Logon JCL.

```
MSGTABC  CSECT                 production M*S*G parms

         DC     A(ACECHRG)     ACE charging parm

         DC     A(FIRSTJ)      first JOBNAME entry
         DC     A(LASTJ)       last JOBNAME entry
         DC     A(FIRSTP)      first TSO pool JOBNAME entry
         DC     A(ACECHRG)     TSO pool CCN Charge Number

         DC     H'1'           sessions at startup
         DC     H'0'           minimum idle sessions
         DC     H'2'           maximum idle sessions
         DC     H'7'           maximum sessions

         DC     A(FIRSTS)      first PROCESS entry
         DC     A(LASTS)       last PROCESS entry


ACECHRG  DC     CL10'ABC123 III'

FIRSTJ   JOBNAME 'ABC123XA'
         JOBNAME 'TSOHCL'
         JOBNAME 'TSOLPR'
         JOBNAME 'TSORTB'
FIRSTP   JOBNAME 'TSOIII'
         JOBNAME 'TSOJJJ'
LASTJ    JOBNAME 'TSOKKK'

FIRSTS   PROCESS 'T#',1,003,'FOREMAN'
         PROCESS 'T#',2,022,'T2'
LASTS    PROCESS 'BX',*,***,'T2BATCH'
```

Figure 8. Sample MSGTABC Assembly.

```
MSGTABS  CSECT                  production MSG parms

         DC     A(FIRSTN)       first CLASS entry
         DC     A(LASTN)        last CLASS entry
         DC     A(FIRSTH)       first HOST entry
         DC     A(LASTH)        last HOST entry
         DC     A(FIRSTP)       first REMOTE entry
         DC     A(LASTP)        last REMOTE entry
         DC     A(RCODETAB)     return code conversion table
         DC     A(CCNENT)       CCN HOST entry

         DC     A(15*6000)      SEQSTRM time-out: 15 min.
         DC     A(00*6000)      HCT     time-out: infinite

         DC     H'5'            process quench limit
         DC     H'4'            process hold limit
         DC     H'8'            process xmit limit
         DC     H'4'            process connection limit


FIRSTN   CLASS  001,'FE'        (generic code, name)
         CLASS  002,'WM'
         CLASS  003,'FOREMAN'
         CLASS  004,'FLPKG'
         CLASS  005,'NFLPKG'
         CLASS  006,'IBS'
         CLASS  007,'WMO'
         CLASS  021,'T1'
LASTN    CLASS  022,'T2'

FIRSTH   HOST   049,049,20447261,HOST10,00  BBN-TENEXB
         HOST   044,044,00000029,HOSTML,00  MIT-MULTICS
CCNENT   HOST   065,065,00000029,HOSTOS,31  UCLA-CCN
LASTH    HOSTN  065,565,00000027,HOSTOS,33  UCLA-CCN-X

FIRSTP   REMOTE CCN,003         (host, generic code)
         REMOTE CCN,004
LASTP    REMOTE CCN,022

         DC     0H'0'           alignment
RCODETAB DC     AL2(HCUNKPRC,BADPRCSS)
         DC     AL2(HCPRCFUL,DESTQFUL)
         DC     AL2(HCGENSPC,BADBITS)
         DC     AL2(HCBADGEN,BADSHORT)
         DC     AL2(HCBADINC,BADHOSTI)
         DC     AL2(0)          end of table
```

Figure 9. Sample MSGTABS Assembly.

REFERENCES

[1] "The National Software Works," Robert E. Millstein, Document No. CADD-7607-3011, Massachusetts Computer Associates, Wakefield, Mass., August 1976.

[2] "The National Software Works," Robert T. Braden, CCN Technical Report TR-9, UCLA Campus Computing Network, August 1976.

[3] "Works Manager Procedures", Chapter 2 in "Semiannual Technical Report," Document No. CADD-7603-0411, Massachusetts Computer Associates, Wakefield, Mass., March 1976.

[4] "An IP Server for NSW-- Semiannual Technical Report," R. T. Braden and H. C. Ludlam. CCN Technical Report TR-7, UCLA Campus Computing Network, UCLA, April 1976.

[5] "A Tool-Bearing Host in the National Software Works," R. T. Braden and H. C. Ludlam, CCN Technical Report TR-10, UCLA Campus Computing Network, UCLA, March 1977.

[6] "MSG: The Interprocess Communication Facility for the National Software Works," NSW protocol Committee, Document No. CADD-7612-2411, Massachusetts Computer Associates, Wakefield, Mass., December 1976 (rev.).

[7] "File Package: The File Handling Facility for the National Software Works," Paul M Cashman, Ross A. Faneuf, and Charles A. Muntz, Document No. CADD-7612-2711, Massachusetts Computer Associates, Wakefield, Mass., December 1976 (rev.).

[8] "The Foreman: Providing the Program Execution Environment for the National Software Works," Richard E. Schantz and Robert E. Millstein. Document No. CADD-7604-0111, Massachusetts Computer Associates, Wakefield, Mass., March 1976.

[9] "Programmers' Guide to the Exchange," Robert T. Braden and Stuart C. Feigin, CCN Technical Report TR-5, UCLA Campus Computing Network, UCLA, March 1971.

[10] "A Server Host System on the ARPANET," Robert T. Braden, Paper in preparation, UCLA Campus Computing Network, UCLA, March 1977.

[11] "Host/Host Protocol for the ARPA Network," Alex Mckenzie, NIC#8246, Network Information Center, Stanford Research Institute, Menlo Park, Calif., January 1972.

[12] "Official Initial Connection Protocol," Jon B. Postel, NIC#7104, Network Information Center, Stanford Research Institute, Menlo Park, Calif., January 1972.

[13] "OS PL/I Checkout and Optimizing Compilers: Language Reference Manual," IBM Corporation, Document GC33-0009.

[14] "Time Sharing Option: Command Language Reference," IBM Corporation, Document GC28-6732.

[15] "ARPA TELNET," Victor Tolomei, CCN Systems Document S-202, UCLA Campus Computing Network, UCLA, April 1977.

[16] "ICT Monitor Services and Macros," Stephen M. Wolfe, Systems document #Q-037, UCLA Campus Computing Network, UCLA, September 1974 (rev.)

[17] "TCAM/2741 Intercept Protocol," Stuart C. Feigin, Systems document #S-112, UCLA Campus Computing Network, August 1972.

[18] "MEMO, An Experimental SVC for Inter-Task Communication," D. Worth and V. Tolomei, Systems document #S-155, UCLA Campus Computing Network, UCLA, January 1974.

[19] "Features of the System Routine SVC," L. P. Rivas, Systems document #S-187, UCLA Campus Computing Network, UCLA, November 1975.

[20] "The Guardian and Service SVC," D. Worth, Systems document #S-136, UCLA Campus Computing Network, UCLA, September 1974.

[21] "ARPA NETWORK Message Processing (Pseudo-)Tasks: Programming Rules," S. M. Wolfe, Systems document #Q-039, UCLA Campus Computing Network, UCLA, July 1973.